The Pragmatic Programmers

Web Development with Clojure

Clojure Web 开发实战

[美] Dmitri Sotnikov 著

张恒 译

中国工信出版集团

人民邮电出版社
POSTS & TELECOM PRESS

The Pragmatic Programmers

Web Development with Clojure

Clojure Web 开发实战

[美] Dmitri Sotnikov 著

张恒 译

人民邮电出版社

北京

图书在版编目（CIP）数据

Clojure Web开发实战 / (美) 肖特尼科夫(Sotnikov,D.) 著 ; 张恒译. -- 北京 : 人民邮电出版社, 2015.11
ISBN 978-7-115-39893-2

Ⅰ. ①C… Ⅱ. ①肖… ②张… Ⅲ. ①程序语言－语言设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第188698号

版权声明

◆ 著　　［美］Dmitri Sotnikov
译　　张　恒
责任编辑　陈冀康
责任印制　张佳莹　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：13.75
字数：262 千字　　2015 年 11 月第 1 版
印数：1－2 500 册　　2015 年 11 月北京第 1 次印刷

著作权合同登记号　图字：01-2014-5439 号

定价：45.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315

内容提要

Clojure 是一门 Lisp 方言。它通过函数式编程技术，直接支持并发软件开发，得到众多开发人员的欢迎，广泛应用于各个领域。Web 开发是 Clojure 的主战场之一。

本书专门探讨 Clojure 在 Web 开发领域的实际应用。通过阅读本书，读者既可以深入理解 Clojure Web 栈的专业知识，同时又能运用这些知识来轻松构建 Web 应用。全书共 7 章，详细介绍了 Clojure Web 开发的各个方面，附录部分介绍了 IDE 的选择、Clojure 快速入门以及相关的数据库技术。

本书适合各个层次的读者。如果具备一些函数式方面的编程经验，将对阅读本书有所助益，但这并不是必需的。如果你还没有真的用过 Clojure，可以快速掌握如何运用这门语言来解决 Web 开发中的实际问题。

对本书的赞誉

这是一本极好的书，我会强烈推荐身边所有的 Clojure Web 开发者阅读。

- Colin Yates，QFI 咨询事务所首席工程师、技术团队领袖

Clojure 这种语言棒极了，用它来开发 Web 应用简直是一种享受。对于诸多熟悉用 Clojure Web 开发工具库的人来说，这本书是份宝贵且及时的材料。

- Fred Daoud，资深 Web 开发者、《七周七 Web 框架》合著者

对于想用 Clojure 开发 Web 应用的人来说，Dmitri Sotnikov 会通过 Clojure 开发实践让你迅速上手。如果你已懂一些 Clojure 基本知识，但是还只在了解的阶段，那这本书太适合你了。

- Russ Olsen，Cognitect 副总裁、咨询服务

Sotnikov 阐明了如何借助 Clojure 的灵活性来搭建网站，使用顶尖的库来制作务实的站点。

- Chris Houser，《Clojure 编程乐趣》合著者

有了这本书，您将手操强大的函数式编程技术直接步入 Web 开发。随着渐学渐长，你还会为你的应用注入更多可扩展、可维护的特性，并且，这种 Clojure 开发经验还会延续到 JavaScript 客户端。

- Ian Dees，《Cucumber Recipes》作者

Dmitri 在书中通过穿插介绍语言特性的同时，成功解决了现在软件开发中遇到的真实问题。这说明你投入时间阅读本书将富有效率和价值。

- Brian Sletten，Bosatsu 咨询、《Resource-Oriented Architecture Patterns for Webs of Data》作者

本书通篇详细介绍用 Clojure 搭建网站应用，节奏轻快、直白。从第 1 章开始，你

将陆续接触真实的 Web 应用，接下来为其添加数据库、处理安全问题、JavaScript 等。不教条、不唠叨、不废话！就为了简洁、高效。本书从无到有提供一个货真价实的应用，使得你可以通过完善不断成长。

- Sam Griffith Jr.，语言专家、任职于 Interactive Web Systems

简 介

这是一株盆景，优雅、高洁。之所以选择盆景作为本书的封面，是因为相同的品质在 Clojure 这门极具魅力的语言身上，同样展现得淋漓尽致。开发软件，就像是修剪一株盆景，只有精工细作，才能将其雕琢成型；也唯有工具趁手，方可体味其中的乐趣。而 Clojure，就是这个不可思议的工具。相信在读完本书之后，你也会这么认为的。

本书适合你吗？

本书适合各个层次的读者。如果具备一些函数式方面的编程经验，将对阅读本书有所助益，但这并不是必需的。如果你还没有真的用过 Clojure，那么阅读本书会是一个不错的起点，因为本书关注的，就是如何运用这门语言来解决实际问题。这意味着，我们仅仅需要少量语言特性，就能实现常见的 Web 应用。

为什么选择 Clojure？

作为一门轻巧的语言，Clojure 的首要目标是简洁并且准确。此外，作为一门函数式语言，它还格外强调不变性（Immutability），以及声明式编程（Declarative Programming）。正如你将在本书中看到的那样，这些特性使得编写清爽又正确的代码，竟会变得如此简单且自然。

编程语言各有千秋，关于它们之间孰优孰劣的争论也从未休止。有的语言，结构简单却表达冗长。也许你曾听人这样说过，表述啰唆一点并没什么大不了，理由是，只要两种语言都是图灵完备的，那么能用简洁语言表达出来的东西，用冗长一些的语言也可以表达出同样的含义，只不过是多出几行代码罢了。

然而，这种说法却忽略了关键所在。因为真正的问题并不在于能不能表达出来，而在于表达得好不好，直不直接。一门好的语言，能让你始终围绕着问题域去思考；而糟糕的语言，则迫使你不得不把问题转换为这门语言强加的概念。

后者，往往都是枯燥无味的。最终，你通篇都是样板代码，并且一再重复着这些早已做过无数遍的事情。如果我们总是不得不编写许多重复的代码，那多少也显得有些可悲了。

还有一些其他语言，它们并不冗长，甚至还提供了诸多用来解决各种问题的工具。可惜的是，绝大多数工具都未必能真正转化为更强的生产力。

语言具备的特性越多，你就越需要花费更多的精力来考虑如何有效地运用这门语言。我之前用过许多这样的语言，发现自己总会费尽心思地纠缠于众多特性之间，难以自拔。

于我而言，何谓理想的语言，就是我可以不假思索地使用它。当一门语言缺乏表达力，就会明显让我感觉到捉襟见肘。另一方面，当一门语言有太多特性的时候，我又会经常感到不知所措，甚至受其所扰。

用数学来进行类比，能记住一个可以推导其他公式的通用公式，总是好过死记硬背一大堆针对特定问题的公式。

Clojure 正是为此而生。它使得我们借助少量的几个通用模式，就可以轻松获得解决特定问题的方案。你只需要学习几个简单的概念，以及些许语法，就可以迅速将它们转化为生产力。这些概念可以用无数种方法加以组合，用来解决任何类型的问题。

为何选 Clojure 来构建 Web 应用？

Clojure 被广泛应用于各个领域，在其数以万计的使用者中，不乏银行和医院这样挑剔的用户。说 Clojure 是 Lisp 语系发展至今最流行的一门方言，也毫不为过了。尽管这门语言还很年轻，但它已经充分证明了自己。这份自信源自于它在生产系统中的表现，也源自于用户们排山倒海般的好评。

由于 Web 开发是 Clojure 的主战场之一，一些重要的库和框架也开始在这个领域崭露头角。一般来说，Clojure 的 Web 栈是基于 Ring[①]和 Compojure[②]的，其中 Ring 是 HTTP 基础库，而 Compojure 则在 Ring 的基础上，提供了路由机制。在接下来的章节中，你将会逐渐地了解并熟悉这个 Web 栈，并懂得如何有效地借助它们来构建你自己的 Web 应用。

① https://github.com/ring-clojure/ring
② https://github.com/weavejester/compojure

目 录

第1章

起步

在简介部分，我们谈到了在编写应用程序时，采用函数式编程风格能够获得诸多好处。当然，想要学会一门语言，仅仅通过阅读是远远不够的，只有亲手编写一些代码，你才能获得真切的体验。

在本章中，我们将会介绍如何开发一个简单的留言簿应用，用户可以使用它给他人留言。通过它，我们能够了解 Web 应用的基本结构，并且尝试一些高效的 Clojure 开发工具。如果你是一个 Clojure 新手，那我建议你先跳到“附录 2 Clojure 入门”，快速了解一下 Clojure 的基本概念和语法。

1.1 环境设置

Clojure 需要 Java 虚拟机（JVM，Java Virtual Machine）才能运行，此外，你还需要一份 1.6 或是更高版本的 Java 开发工具包[①]（JDK，Java Development Kit）用于开发。Clojure 是作为一个 JAR 包来分发的，你只需简单地将其包含在工程的 class-path 中即可。你可以使用任何常规的 Java 工具来构建 Clojure 应用，比方说 Maven[②]或者 Ant[③]。不过，我强烈建议你使用 Leiningen[④]，它是专为 Clojure 定制的。

① http://www.oracle.com/technetwork/java/javase/downloads/index.html
② http://maven.apache.org/
③ http://ant.apache.org/
④ http://leiningen.org/

使用 Leiningen 管理工程

借助 Leiningen，你可以建立、构建、测试、打包和部署工程。也就是说，它能为你提供工程管理方面的一站式服务。

Maven 是一个非常流行的 Java 依赖关系管理工具，而 Leiningen 就相当于 Clojure 世界中的 Maven。重点是，Leiningen 与 Maven 兼容，因此它可以毫无障碍地访问那些得到精心维护，且存放着海量 Java 类库的存储中心。此外，Clojure 的库通常可以在 Clojars[①]这个存储中心找到。所以，默认情况下 Leiningen 是启用了 Clojars 的。

使用 Leiningen，你不用手动去下载那些在工程中需要用到的库。你只需要简单地声明一下工程的顶级依赖，剩下的事情 Leiningen 就会帮你自动搞定。

Leiningen 的安装实在是小菜一碟，只需要从官方主页[②]上下载并执行安装脚本即可。

不如动手试试看。我们会通过执行下列命令，来下载这个脚本，并创建一个全新的 Clojure 工程：

```
wget https://raw.github.com/technomancy/leiningen/stable/bin/lein
chmod +x lein
mv lein ~/bin
lein new myapp
```

由于这是我们第一次运行 lein 这个命令，它做的第一件事情是安装它自己。一切顺利的话，你将会看到下面的输出：

```
Generating a project called myapp based on the 'default' template.
To see other templates (app, lein plug-in, etc), try `lein help new`.
```

一个新的文件夹 myapp 就创建好了，里面是应用程序的骨架。应用程序的代码存放在 src 文件夹中。其中有另外一个 myapp 文件夹，这个文件夹中只有一个文件，名为 core.clj。文件内容如下：

① https://clojars.org/

② http://leiningen.org/#install

```
(ns myapp.core)

(defn foo
  "I don't do a whole lot."
  [x]
  (println x "Hello, World!"))
```

请注意命名空间的声明，与其文件夹结构是相匹配的。由于命名空间 core 位于 myapp 目录当中，所以它的名字就是 myapp.core。

Leiningen 工程文件一瞥

在工程文件夹 myapp 里有一个 project.clj 文件。这个文件包含了应用程序的描述信息，你可以仔细观察一下，就会发现这个文件是用标准的 Clojure 语法编写的，描述了应用的名称、版本、网址、许可证信息和依赖项，如下所示。

```
(defproject myapp "0.1.0-SNAPSHOT"
:description "FIXME: write description"
:url "http://example.com/FIXME"
:license {:name "Eclipse Public License"
          :url "http://www.eclipse.org/legal/epl-v10.html"}
:dependencies [[org.clojure/clojure "1.5.1"]])
```

通过修改这个 project.clj 文件，能让我们控制应用程序的方方面面。例如，我们可以通过添加:main 关键字，将 myapp.core 命名空间下的 foo 函数设置为应用的入口点：

```
(defproject myapp "0.1.0-SNAPSHOT"
:description "FIXME: write description"
:url "http://example.com/FIXME"
:license {:name "Eclipse Public License"
          :url "http://www.eclipse.org/legal/epl-v10.html"}
:dependencies [[org.clojure/clojure "1.5.1"]]
;;this will set foo as the main function
:main myapp.core/foo)
```

此时我们就可以通过执行 lein run 这个命令来运行应用了。由于 foo 函数要求传入一个参数，我们只得遵命行事：

```
lein run First
First Hello, World!
```

在前面这个例子中，我们创建的应用非常简单，只有一个依赖项：Clojure 运行时。如果我们直接以此为基础来开发 Web 应用的话，就免不了要编写大量的样板代码，才能让它运行起来。下面就让我们看看如何利用 Leiningen 的模板，来创建一个开箱即用的 Web 应用吧。

Leiningen 的模板

当把模板的名称提供给 lein 脚本时，就可以根据其对应的模板来初始化工程骨架。其实模板自身也不过是使用了 lein-newnew 插件[①]的 Clojure 工程罢了。稍后我们将看到如何创建自己的模板。

眼下，我们将会使用 compojure-app 模板[②]来初始化下一个应用。执行 lein 脚本时，模板的名称是作为参数传给 new 关键字的，紧接其后的是工程名称。为了创建一个 Web 应用，而不是之前那样的默认工程，我们只需执行以下命令即可：

```
lein new compojure-app guestbook
```

这样 Leiningen 就知道创建留言簿应用时，应该使用 compojure-app 模板了。此类应用需要启动一个 Web 服务才能运行。其实这很容易，我们只需要使用 lein ring server 来替代 lein run 即可。

当我们运行这个应用时，控制台会输出如下信息，与此同时还会弹出一个打开了应用主页的浏览器窗口。

```
lein ring server
guestbook is starting
2013-07-14 18:21:06.603:INFO:oejs.Server:jetty-7.6.1.v20120215
2013-07-14 18:21:06.639:INFO:oejs.AbstractConnector:
StartedSelectChannelConnector@0.0.0.0:3000
Started server on port 3000
```

喔，现在我们已经知道如何创建和运行应用了，接下来不妨考虑一下应该选用什么样的编辑器。

你多半已经留意到，Clojure 代码中有大量的括号。保持它们起止对应很快就会

① https://github.com/Raynes/lein-newnew
② https://github.com/yogthos/compojure-template

成为一种挑战，所幸 Clojure 编辑器会替我们收拾这个摊子，否则会令人产生严重的挫败感。

事实上，这些编辑器不仅仅能平衡括号，其中的一些甚至能够感知其结构。这就意味着编辑器能够理解一个表达式是从什么地方开始，又到什么地方结束的。因此，我们可以根据逻辑上的代码块来导航和选取，而非简单针对文本行号。

在本章中，我们将会选用 Light Table[①]来开发留言簿应用。获取并运行 Light Table 是非常容易的，这样我们就能尽快投入到代码的编写中了。然而，它的功能还比较有限，在较大的工程中，你对此可能有较深的体会。“附录 1 选择 IDE”中还有对其他开发环境的讨论。

使用 Light Table

Light Table 不需要安装，下载完成后即可直接运行。

Light Table 的外观相当简洁。默认情况下，它仅在编辑器窗格中显示了几行欢迎信息，如图 1-1 所示。

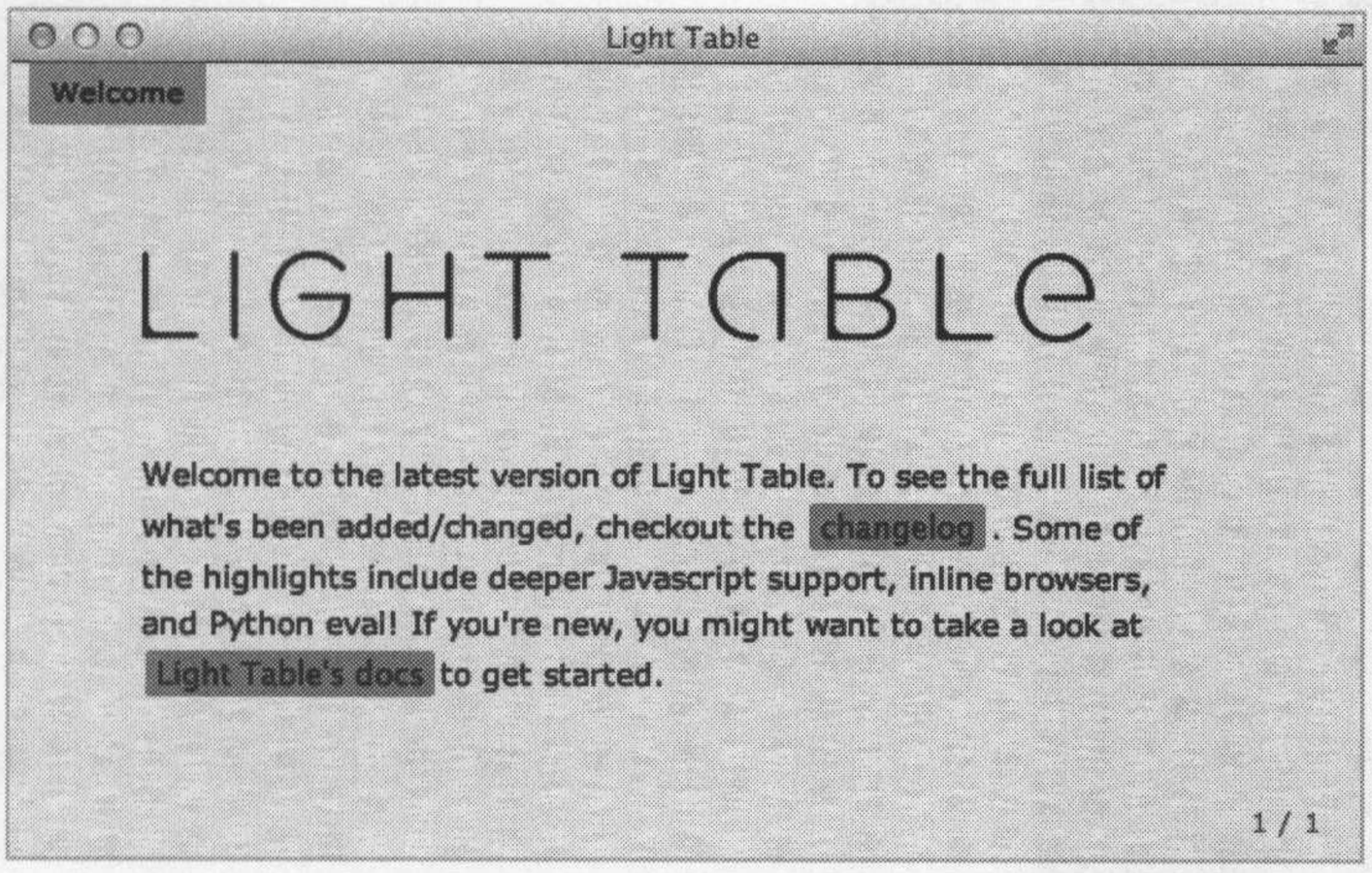

图 1-1 Light Table 工作区

为了显示 workspace 面板，我们可以在菜单中选择 View →Workspace，或是按下 Ctrl+T（Windows/Linux）组合键或 Cmd+T（OS X）组合键。

① http://www.lighttable.com/

如图 1-2 所示，我们可以在 workspace 的 folder 标签页中打开留言簿工程。

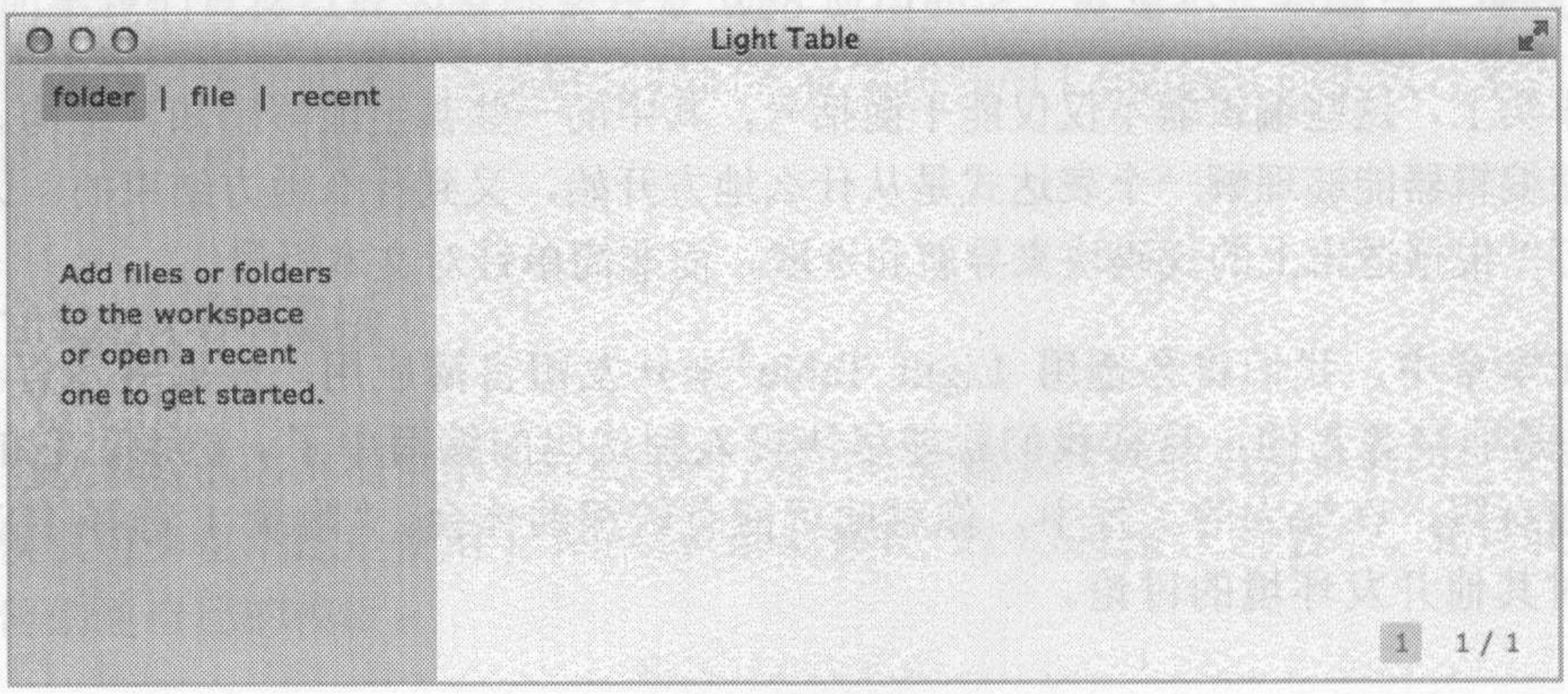

图 1-2　打开工程

一旦工程被选中，我们就可以浏览整个工程树，并选择我们想要编辑的文件，如图 1-3 所示。

现在，开发环境已经就绪，看起来我们终于可以为留言簿应用添加一些功能了。

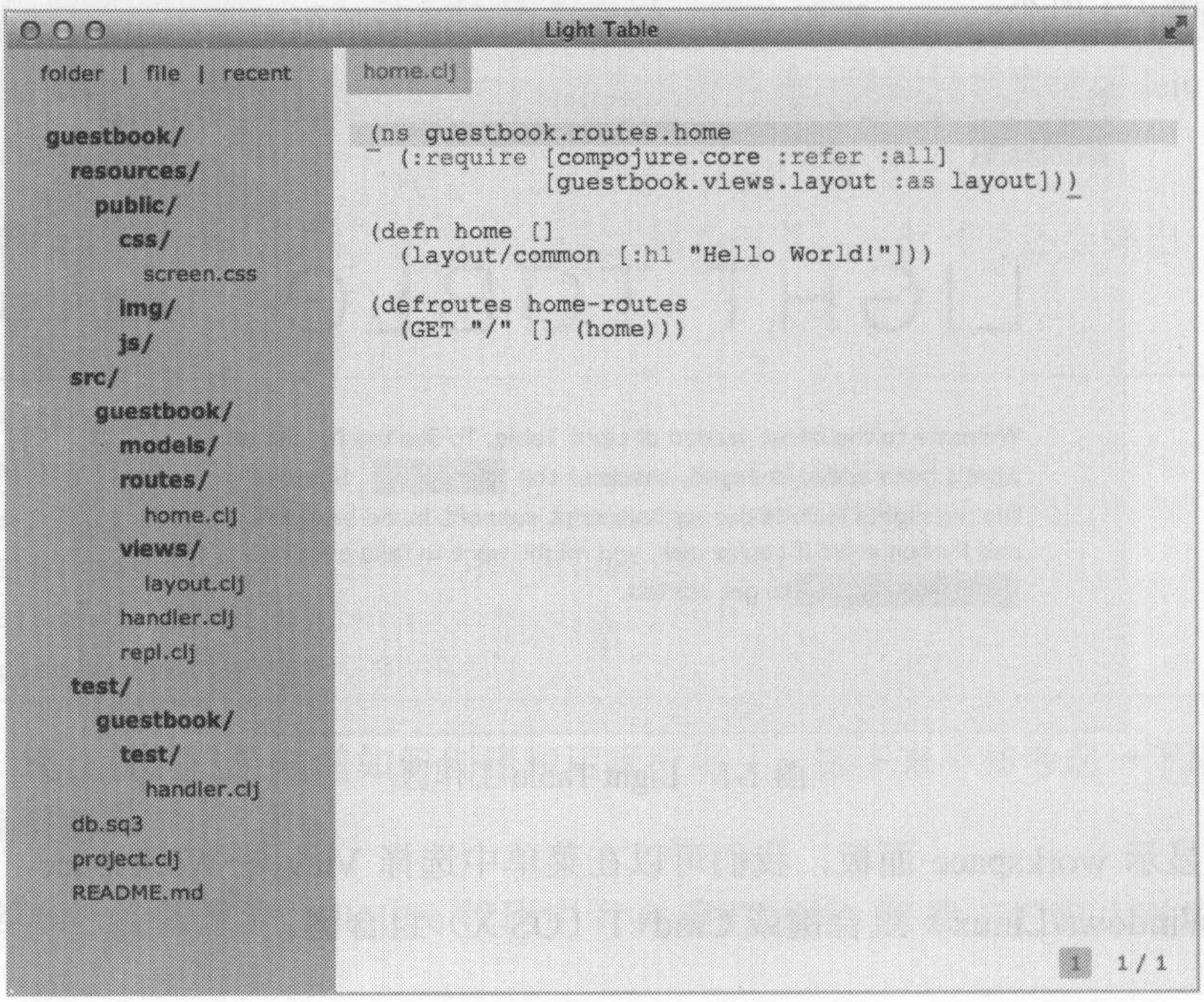

图 1-3　Light Table 的工程

1.2 你的第一个工程

你的留言簿应该已经在控制台运行了，可以通过 http://localhost:3000/来访问。在控制台终端按下 Ctrl+C，就能停止它的运行。既然我们已经在 Light Table 的工作区打开了这个工程，不妨就直接在编辑器中运行它吧。

我们现在要更进一步，创建一个“读取—求值—打印循环”（REPL，Read-Evaluate-Print Loop），将 Light Table 连接至我们的工程。菜单 View →Connections 可以打开连接标签页。如图 1-4 所示，让我们点击标签页中的 Add Connection 按钮。

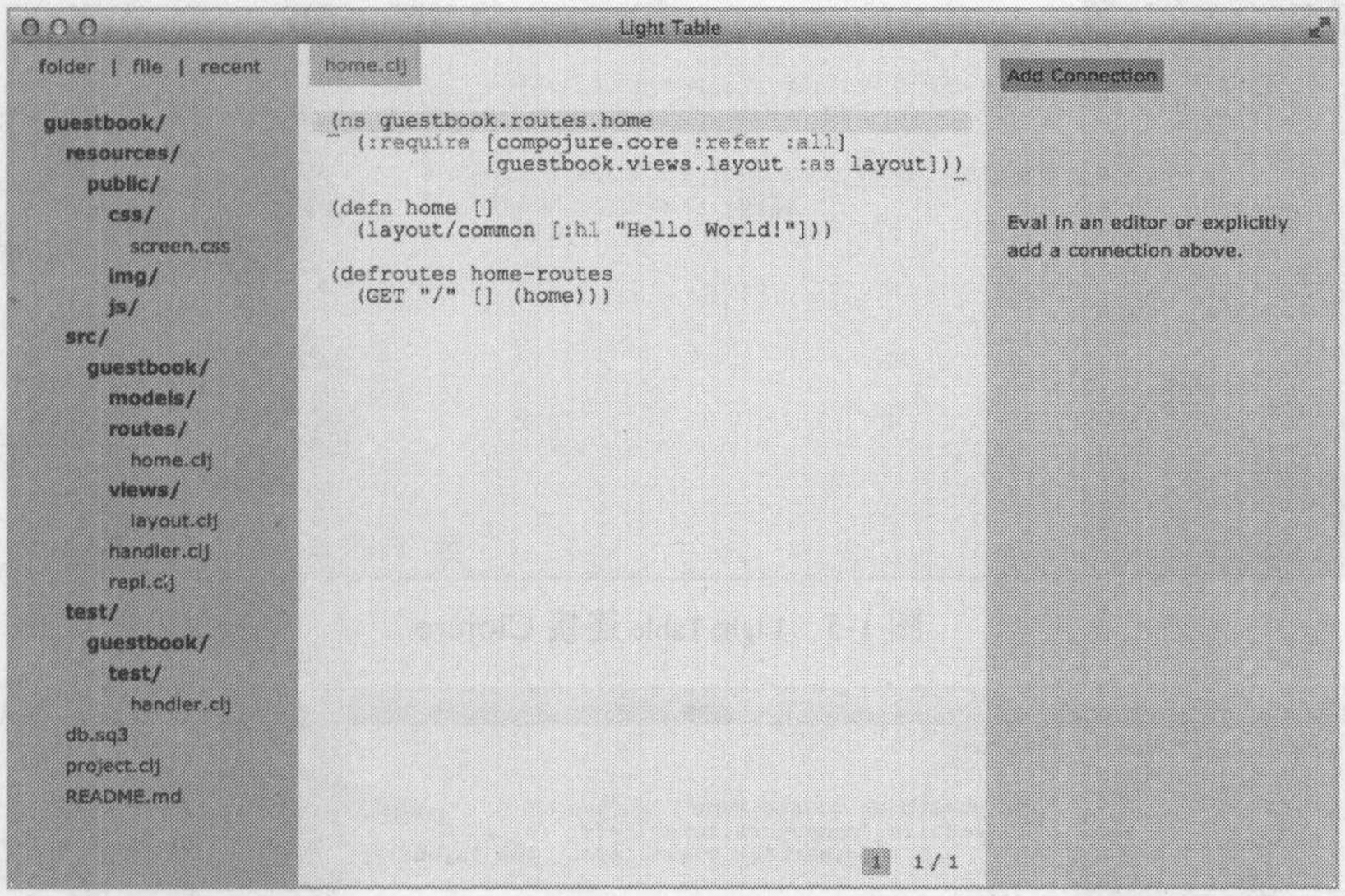

图 1-4 Light Table 的连接

此时，会弹出一个列表，列出了几种不同的连接选项。如图 1-5 所示，接下来选择 Clojure。然后，让我们找到留言簿工程所在的文件夹，并且选中 project.clj 文件。

一旦我们的工程与 Light Table 建立了连接，我们就可以直接在编辑器中对代码进行求值了。

说不如做，你可以立刻挑选一个函数，然后按下 Ctrl+Enter（Windows/Linux）组合键或是 Cmd+Enter（OS X）组合键。如果我们选择的是 home 函数，那么打印出来的内容应该是这样：

```
#'guestbook.routes.home/home
```

这意味着这个函数已经在 REPL 中进行了求值，随时可用了。

另外，按下 Ctrl+Spacebar 组合键后输入 repl，就能打开一个即时 repl。在这个新打开的编辑器窗格中，我们可以随意运行任何代码，如图 1-6 所示。

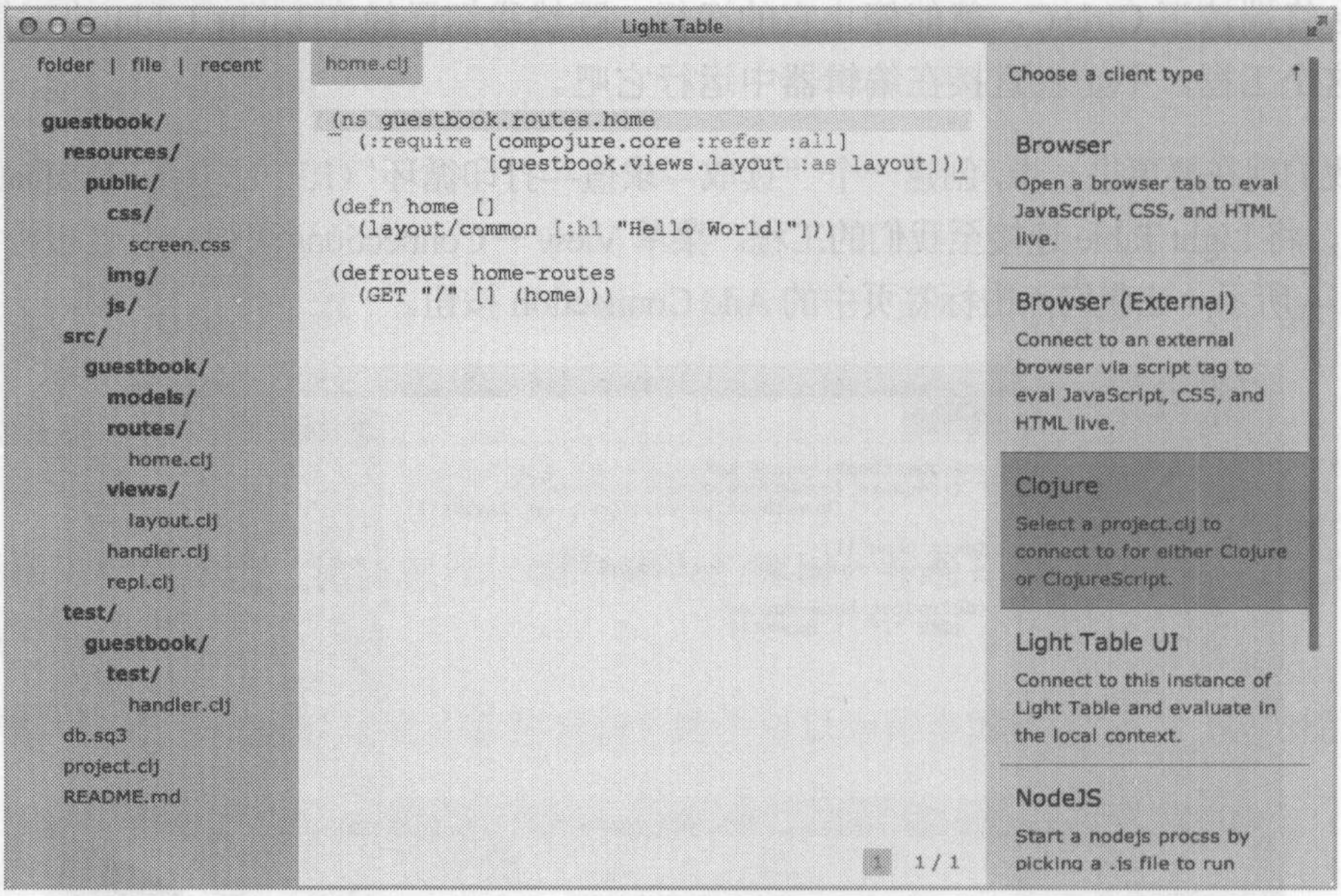

图 1-5　Light Table 连接 Clojure

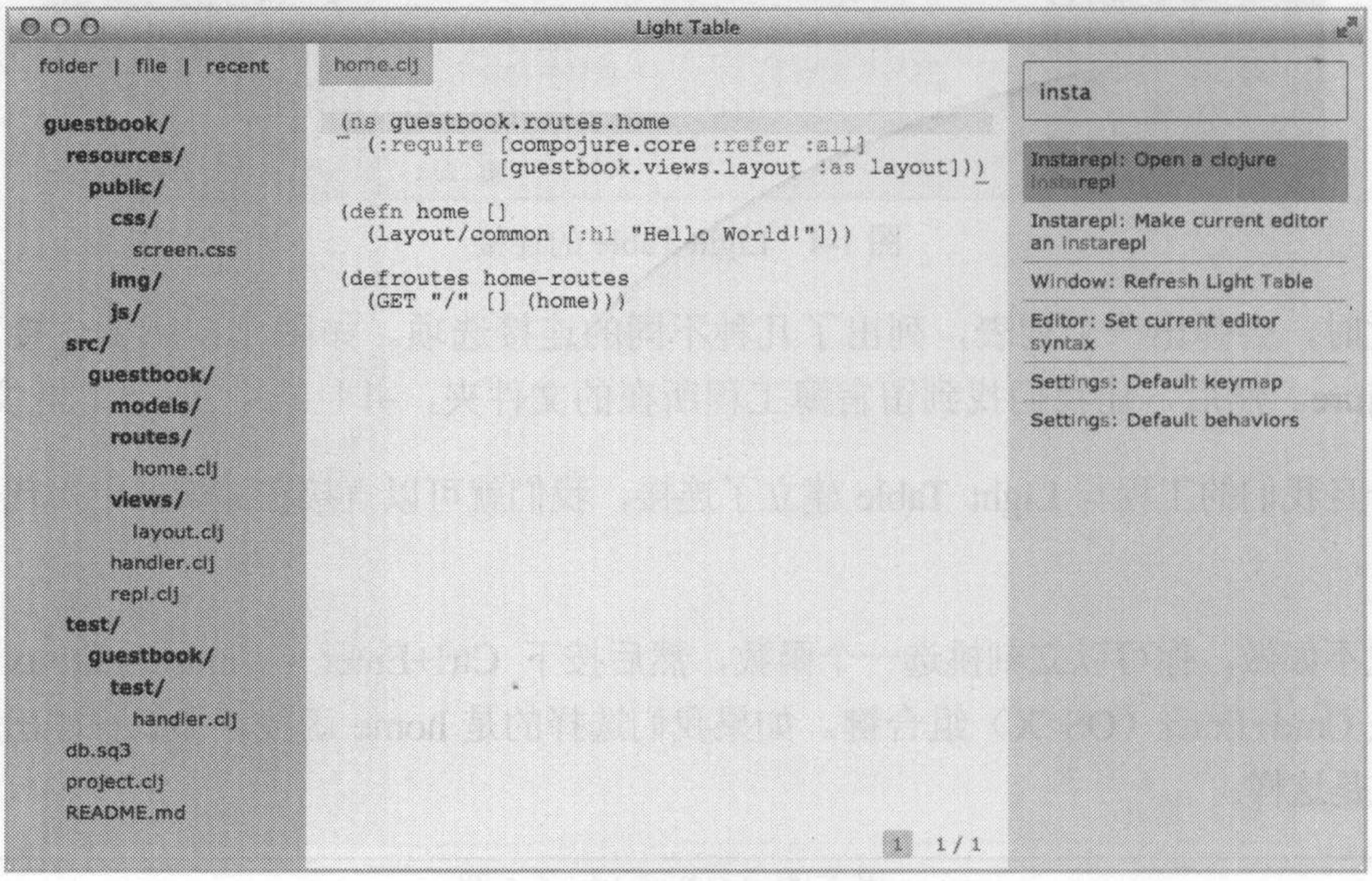

图 1-6　Light Table 的即时 repl

默认情况下，一旦进行任何修改，都会使得即时 repl 中的所有内容被重新求值。这被称为live实时模式。现在，让我们载入guestbook.repl命名空间，然后执行start-server函数。

```
(use 'guestbook.repl)
(start-server)
```

一旦上述代码完成求值，就会启动 HTTP 服务，同时打开一个新的浏览器窗口，指向了应用的主页，如图 1-7 所示。

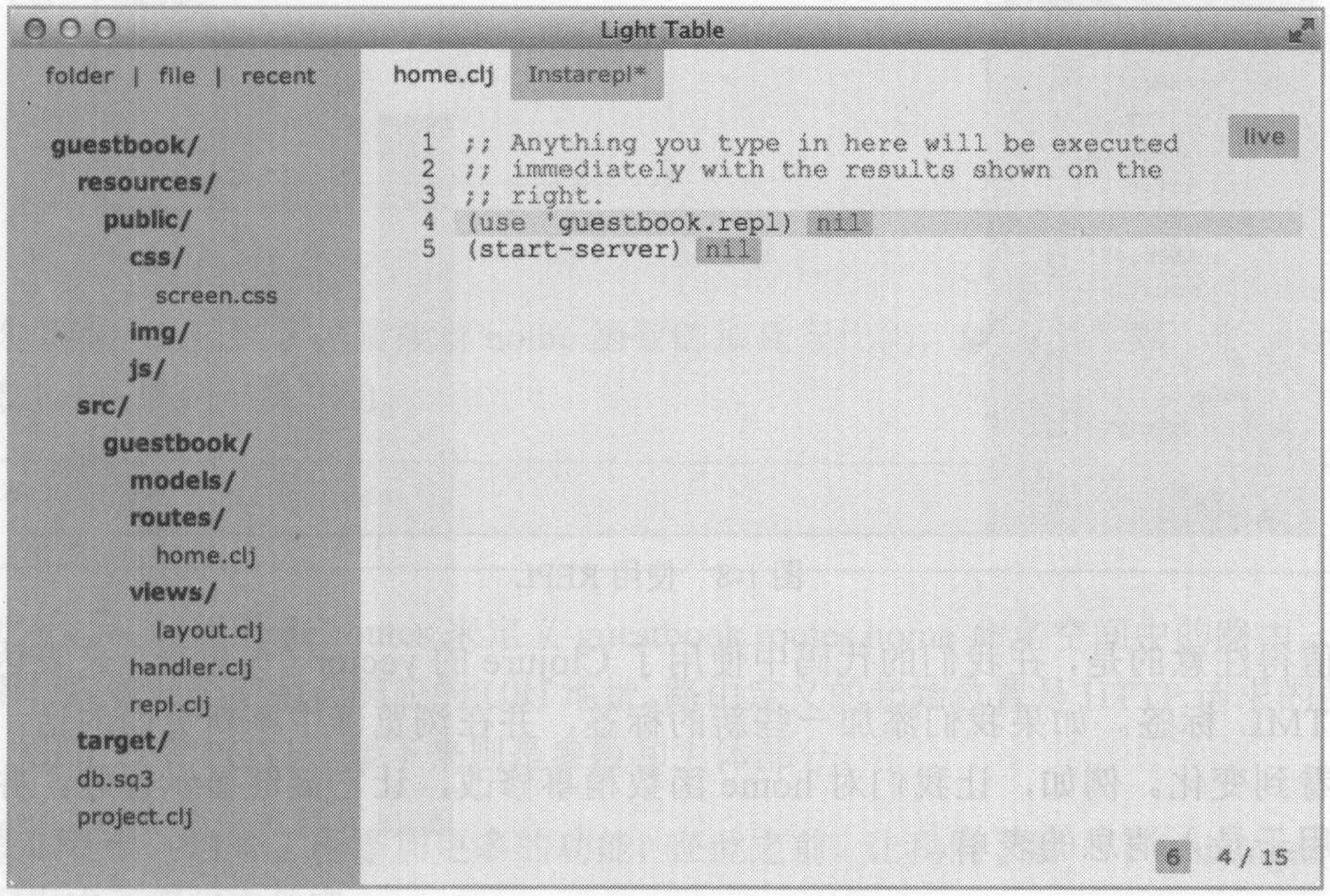

图 1-7 在即时 repl 中运行服务

显然我们不希望 start-server 被反复调用，因此记得从即时 repl 里删除之前的代码。

另外，我们还可以关闭实时求值功能，只要点击右上角的 live 图标即可。禁用了实时模式后，我们可以通过 Alt-Enter 来进行选择性的求值。

下面，如图 1-8 所示，让我们执行(use 'guestbook.routes.home)来导入 home 命名空间，然后调用 home 函数。

如你所见，对 home 的调用只是简单生成了我们的 HTML 主页，一个字符串。这就是我们访问 http://localhost:3000 时，浏览器为我们呈现出来的内容。

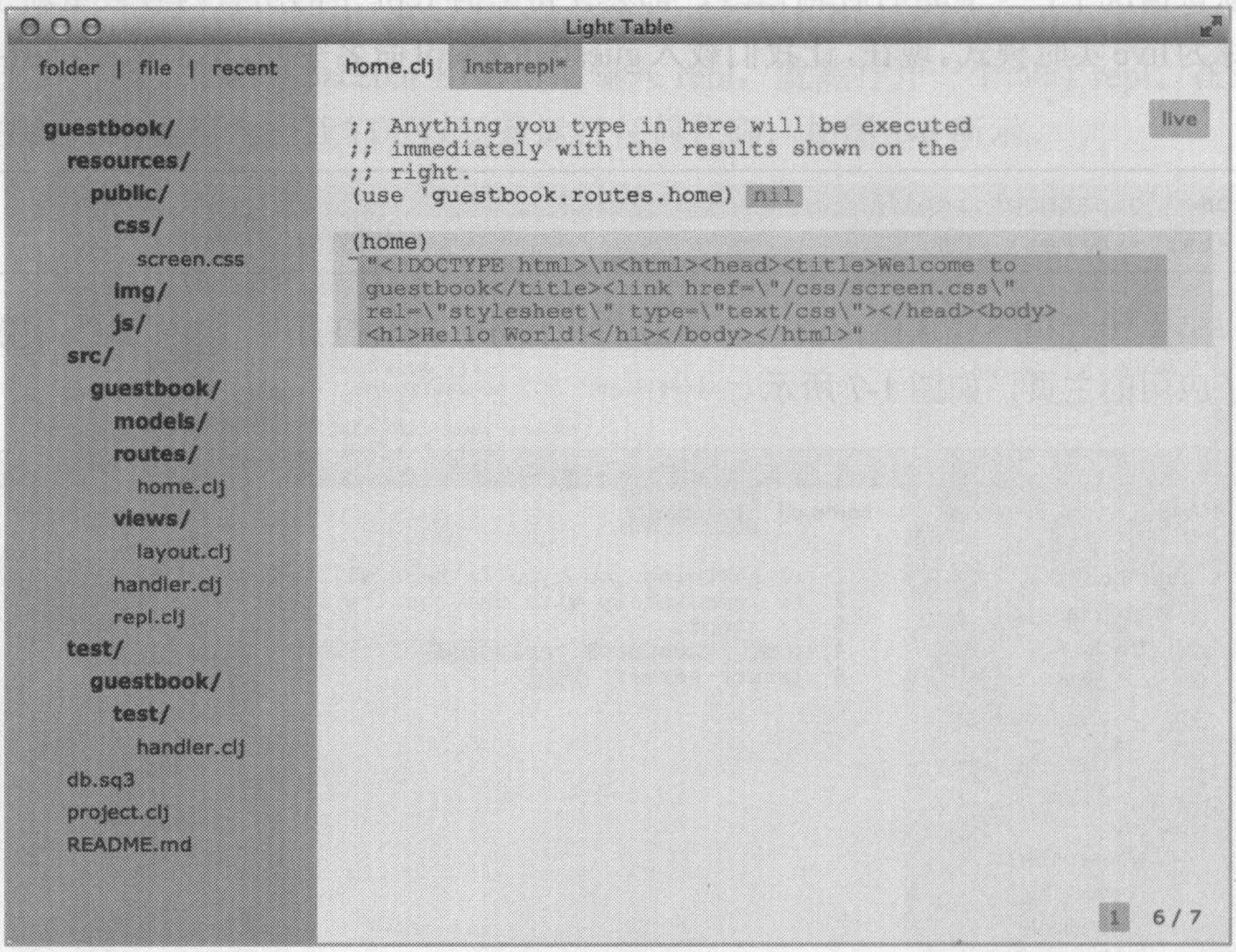

图 1-8　使用 REPL

值得注意的是，在我们的代码中使用了 Clojure 的 vector（矢量表）来表达相应的 HTML 标签。如果我们添加一些新的标签，并在浏览器中刷新页面的话，立刻就能看到变化。例如，让我们对 home 函数稍事修改，让它能够显示标题，并提供一个用于录入消息的表单。

```
(defn home []
  (layout/common
   [:h1 "Guestbook"]
   [:p "Welcome to my guestbook"]
   [:hr]
   [:form
    [:p "Name:"]
    [:input]
    [:p "Message:"]
    [:textarea {:rows 10 :cols 40}]]))
```

好了，刷新一下页面，看到变化了吧，如图 1-9 所示。

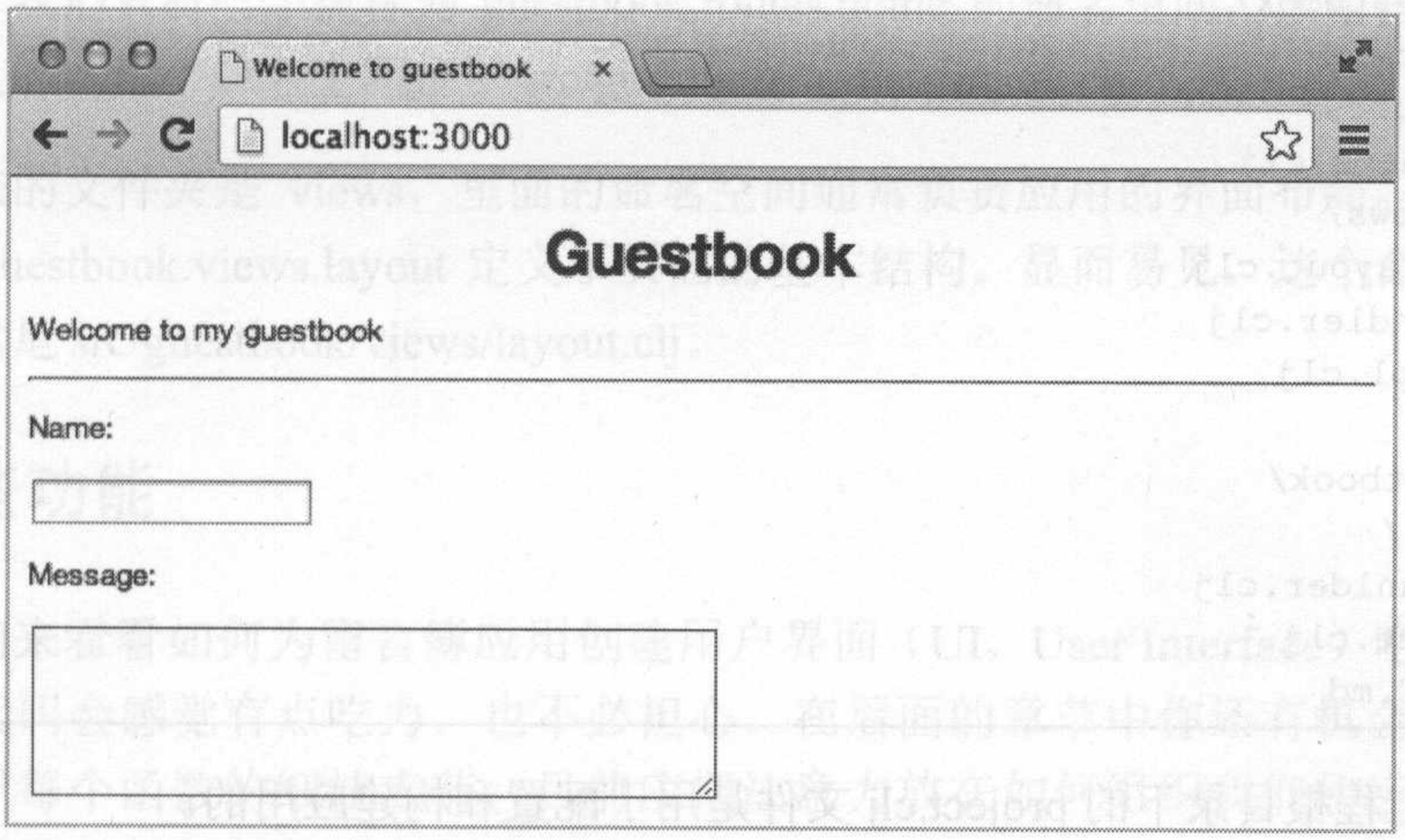

图 1-9 留言簿

你可能已经猜到了，紧接着 home 函数的那几行代码，就是负责将“/”路由和处理函数 home 绑到一块儿的。

```
(defroutes home-routes
  (GET "/" [] (home)))
```

此处，我们使用 defroutes 来定义 guestbook.routes.home 命名空间中的路由。每个路由都代表着一个应用会响应的 URI 地址。路由定义的起始位置是 HTTP 请求的类型，例如，GET 或者 POST，接下来则是参数和主体部分。

我们还会为这个工程添加更多的功能，在此之前，让我们了解一下 Leiningen 模板为我们生成了哪些文件吧。

了解应用程序的结构

在 Workspace 标签页中展开我们的工程之后，看上去应该是这样的：

```
guestbook/
  resources/
    public/
      css/
        screen.css
      img/
      js/
  src
```

```
    guestbook/
    models/
    routes/
      home.clj
    views/
      layout.clj
    handler.clj
    repl.clj
  test/
    guestbook/
    test/
      hanlder.clj
  project.clj
  README.md
```

位于工程根目录下的 project.clj 文件是用于配置和构建应用的。

还有几个文件夹，src 用来存放应用的代码。resources 文件夹则用来存放与应用相关的静态资源，比如 CSS、图片和 JavaScript 脚本。最后，在 test 文件夹中，我们可以为应用添加一些测试。

Clojure 命名空间遵循 Java 的打包约定，也就是说，如果命名空间包含前缀，则其存放的文件夹路径必须与前缀相匹配。需要注意的是，如果一个命名空间包含“-”，则体现在文件夹路径和文件名上时，“-”必须转换为“_”。

这是因为 Java 的包名中不允许出现“-”。而 Clojure 代码最终会被编译为 JVM 字节码，所以也必须遵守这个规则。

由于我们把自己的应用叫作 guestbook，因此它所有的命名空间都被放置在了 src/guestbook 文件夹下。让我们看看都有些什么吧。首先，我们在 src/guestbook/handler.clj 文件中找到了 guestbook.handler 命名空间。这个命名空间包含了应用程序的入口点，此外还定义了被用来处理所有请求的 handler。

在 src/guestbook/repl.clj 文件中的是 guestbook.repl 命名空间，调用里面的函数，就可以在 REPL 中启动和停止服务。我们可以借助它直接从编辑器中启动我们的应用，而不必非得通过 lein 来运行。

接下来，我们有一个名为 models 的文件夹。这是留给应用的模型层的。里面的命名空间也负责连接数据库、定义表结构，还有访问记录等。

在 routes 文件夹下，是那些负责定义路由的命名空间。这些路由构成了我们将要实现的工作流的入口点。

目前，我们只有一个被称为 guestbook.routes.home 的命名空间，应用的主页就是在这里定义的。这个命名空间位于 src/guestbook/routes/home.clj 文件中。

接下来的文件夹是 views，里面的命名空间通常负责应用的界面布局。其自带的命名空间 guestbook.views.layout 定义了页面的基本结构。显而易见，这个命名空间对应的文件就是 src/guestbook/views/layout.clj。

添加一些功能

让我们来看看如何为留言簿应用创建用户界面（UI，User Interface）吧。即使你阅读这些代码会感觉有点吃力，也不必担心，在后面的章节中你还有机会弄明白。相比纠缠于每个函数的细枝末节，目前应把注意力放在如何组织我们的应用，以及如何拆分应用逻辑更为重要。

在前面，我们曾经用纯手工的方式创建了一个录入表单。现在，我们打算用一个更好的实现来替代它，这会用到 Hiccup[①]库提供的辅助函数。

为了使用这些函数，需要把库导入我们的命名空间，像下面这样修改命名空间的声明：

```
(ns guestbook.routes.home
  (:require [compojure.core :refer :all]
            [guestbook.views.layout :as layout]
            [hiccup.form :refer :all]))
```

首先我们创建一个函数，用来呈现已有的消息。这个函数会生成一个包含了现有消息的 HTML 列表。就目前来说，我们先简单地硬编码几条消息就行。

```
(defn show-guests []
 [:ul.guests
  (for [{:keys [message name timestamp]}
        [{:message "Howdy" :name "Bob" :timestamp nil}
         {:message "Hello" :name "Bob" :timestamp nil}]]
    [:li
     [:blockquote message]
     [:p "-" [:cite name]]
     [:time timestamp]])])
```

接下来，我们对 home 函数进行调整，使顾客可以看到前面那些顾客留下的消息。当然，还得提供一个表单用来创建新的消息。

① https://github.com/weavejester/hiccup

```
(defn home [& [name message error]]
 (layout/common
   [:h1 "Guestbook"]
   [:p "Welcome to my guestbook"]
   [:p error]
   ;here we call our show-guests function
   ;to generate the list of existing comments
   (show-guests)
   [:hr]
   ;here we create a form with text fields called "name" and "message"
   ;these will be sent when the form posts to the server as keywords of
   ;the same name
   (form-to [:post "/"]
     [:p "Name:"]
     (text-field "name" name)
     [:p "Message:"]
     (text-area {:rows 10 :cols 40} "message" message)
     [:br]
     (submit-button "comment"))))
```

切换到浏览器，可以看到两条测试消息连同表单一块儿都显示出来了。请留意，现在 home 函数多了几个可选参数。我们会把这些参数的值显示到页面上。如果这些参数为 nil，那么在进行显示时，会把它们视作空字符串。

我们创建的这个表单会向“/”发送 HTTP 的 POST 请求，所以我们再添加一个路由来处理它吧：这个路由将会调用一个名为 save-message 的辅助函数，我们稍后会给出其定义。

guestbook/src/guestbook/routes/home.clj

```
(defroutes home-routes
(GET "/" [] (home))
(POST "/" [name message] (save-message name message)))
```

save-message 函数会检查 name 和 message 这两个参数，然后就去调用 home 函数。倘若两个参数都没问题，那么消息会被打印到控制台；否则，将会生成一条出错信息。

```
(defn save-message [name message]
 (cond
   (empty? name)
   (home name message "Some dummy forgot to leave a name")
   (empty? message)
   (home name message "Don't you have something to say?")
   :else
   (do
     (println name message)
     (home))))
```

来，在留言簿中留一次言试试看，你会看到名字和消息在控制台里打印出来了。接下来，将 name 或者 message 留白，看看有没有显示出错消息。

现在，视图部分已经具备了通过 UI 显示和提交消息的能力。但此时此刻，我们还没有能真正存放这些消息的地方。

添加数据模型

既然我们的应用需要保存访客们的留言，那我们在 project.clj[①]文件中加入对 JDBC 和 SQLite 的依赖项吧。添加完毕后的，:dependencies 看起来应该是下面这样子的：

```
:dependencies [[org.clojure/clojure "1.5.1"]
               [compojure "1.1.5"]
               [hiccup "1.0.4"]
               [ring-server "0.3.0"]
               ;;JDBC dependencies
               [org.clojure/java.jdbc "0.2.3"]
               [org.xerial/sqlite-jdbc "3.7.2"]]
```

因为添加了新的依赖项，我们需要将工程与 REPL 重新连接。首先打开 Connect 标签页并且点击 disconnect 按钮，然后按照先前介绍过的步骤来连接一个新的 REPL 实例，如图 1-10 所示。

一旦重新连上了 REPL，我们就需要在即时 repl 中执行(start-server)，早些时候我们曾经做过一次，还记得吗？

OK，万事俱备，只欠数据模型了。我们会在 src/guestbook/models 文件夹下创建一个新的命名空间。我们把这个命名空间称为 guestbook.models.db。具体做法是：在工作区中，右键单击 models 文件夹，并且选择 New File 选项，然后将这个文件命名为 db.clj。

正如其名称所暗示的，db 命名空间将负责应用的数据模型，并且提供从数据库读取或是向数据库写入数据的功能。

首先，我们需要添加命名空间声明，以及导入数据库依赖项。下面是这个命名空间的声明：

guestbook/src/guestbook/models/db.clj

```
(ns guestbook.models.db
  (:require [clojure.java.jdbc :as sql])
  (:import java.sql.DriverManager))
```

① http://www.sqlite.org/

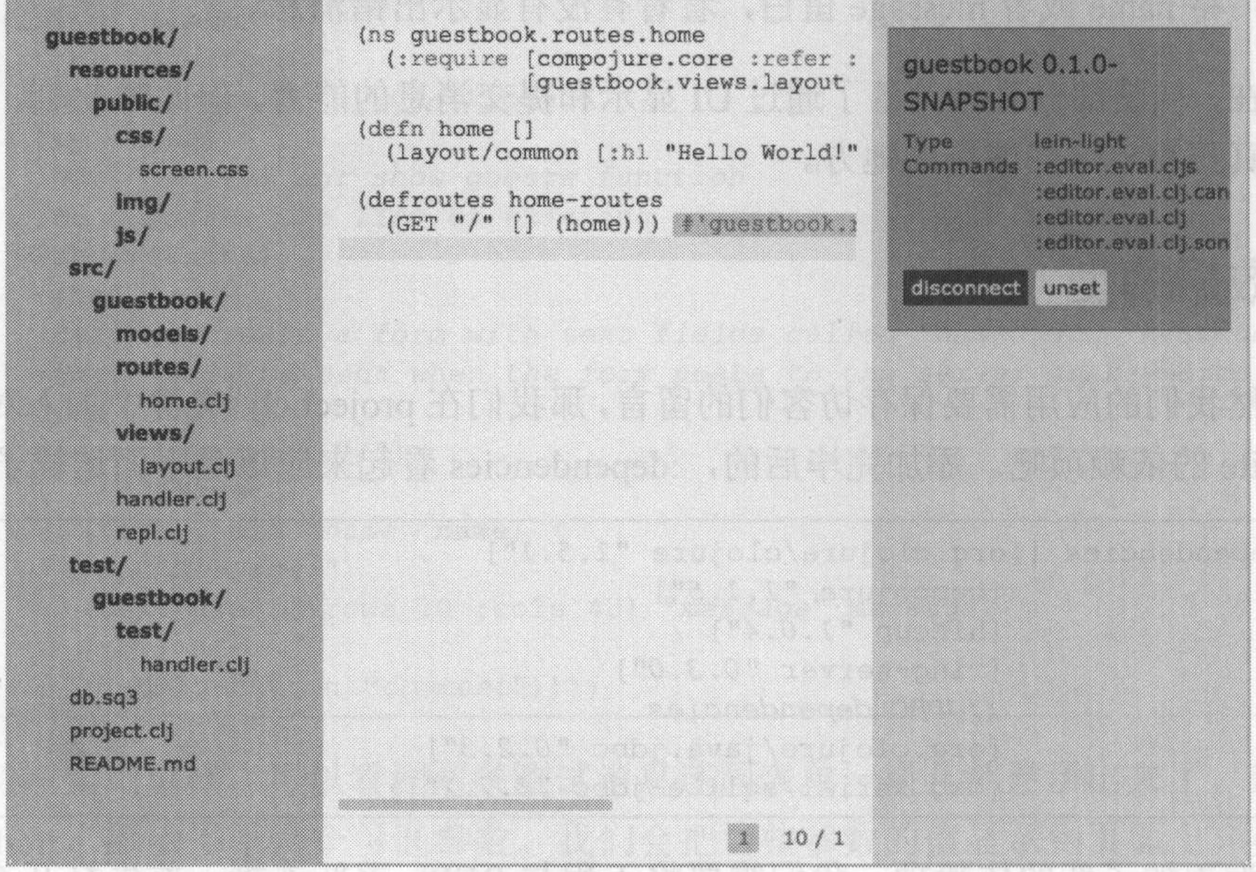

图 1-10　断开 REPL

请注意，导入其他 Clojure 命名空间时，我们使用的是:require 关键字，而导入 Java 类时，我们则用了:import。

下一步，我们将要创建数据库连接的定义。这个定义其实就是一个简单的 map，包含了 JDBC 驱动的类型、协议，以及 SQLite 数据库的文件名。

guestbook/src/guestbook/models/db.clj

```
(def db {:classname   "org.sqlite.JDBC",
         :subprotocol   "sqlite",
         :subname       "db.sq3"})
```

声明了数据库连接之后，我们还需要编写一个函数，创建用于保存访客留言的数据表。

guestbook/src/guestbook/models/db.clj

```
(defn create-guestbook-table []
  (sql/with-connection
    db
    (sql/create-table
```

```
  [:timestamp "TIMESTAMP DEFAULT CURRENT_TIMESTAMP"]
  [:name "TEXT"]
  [:message "TEXT"])
 (sql/do-commands "CREATE INDEX timestamp_index ON guestbook (timestamp)")))
```

这个函数使用了 with-connection 语句，这样就能确保数据库连接在使用完毕后能够得到恰当的清理。在其内部，我们调用 create-table 函数来创建数据表，表名用关键字表示，而表示字段则使用了 vector。为了完整起见，我们还为 timestamp 字段创建了索引。

在即时 repl 中执行(create-guestbook-table)之前，我们首先要导入它的命名空间，前面我们曾经对 guestbook.routes.home 也这么做过，还记得吗？

```
(use 'guestbook.models.db)

(create-guestbook-table)
```

现在你就可以在即时 repl 中执行 create-guestbook-table，把数据表给创建出来了。但有一点需要注意，如果你开启了实时模式，那么最好将其禁用；否则每次即时 repl 临时缓冲的变动，都会导致 create-guestbook-table 被调用并产生错误。

数据表创建完毕，接下来我们就可以编写从数据库中读取留言的函数了。

guestbook/src/guestbook/models/db.clj

```
(defn read-guests []
  (sql/with-connection
   db
   (sql/with-query-results res
     ["SELECT * FROM guestbook ORDER BY timestamp DESC"]
     (doall res))))
```

此处我们使用 with-query-results 与来执行 select 语句，并返回其结果。之所以要在返回之前调用 doall，是因为 res 是惰性的，不会把所有结果都加载到内存中。

通过调用 doall，我们强制对 res 进行了完全求值。如果不这么做的话，一旦离开了函数的作用范围，我们的数据库连接就会被关闭，于是便无法在函数之外访问结果数据了。

我们还需要创建另外一个函数，用来把消息保存到留言簿的数据表中。这个函数会调用 insert-values，并且接受访客的名字和消息作为参数进行保存。

guestbook/src/guestbook/models/db.clj

```
(defn save-message [name message]
  (sql/with-connection
```

```
    db
    (sql/insert-values
      :guestbook
      [:name :message :timestamp]
      [name message (new java.util.Date)])))
```

用于读取和保存消息的函数已经写好，现在我们可以在 REPL 中尝试一下了。我们需要在即时 repl 中重新执行一遍(use 'guestbook.models.db)，这样才能访问这几个新添加的函数。然而，在 guestbook.models.db 和 guestbook.routes.home 这两个命名空间中都定义了名为 save-message 的函数。

如果尝试重新加载 guestbook.models.db 命名空间，我们会得到一个错误，指出 save-message 已经从 guestbook.routes.home 命名空间导入过了。为了避免这个问题，在执行(use 'guestbook.models.db)之前，我们需要在即时 repl 中先执行 ns-unmap，移除当前对 save-message 的引用。

```
(ns-unmap 'user 'save-message)
(use 'guestbook.models.db)
```

现在我们可以尝试运行下面的代码，看看保存和读取消息的逻辑是否符合预期：

```
(save-message "Bob" "hello")
(read-guests)
```

将留言保存到数据库，然后读取出来以后，我们应该能看到图 1-11 所示的输出。

有了持久层，我们就可以回过头去修改 home 命名空间，将先前那些硬编码的假数据统统扔掉了。

组合起来

现在我们可以把对 db 的依赖项添加到 home 路由的命名空间声明中了。

```
guestbook/src/guestbook/routes/home.clj
(ns guestbook.routes.home
  (:require [compojure.core :refer :all]
            [guestbook.views.layout :as layout]
            [hiccup.form :refer :all]
            [guestbook.models.db :as db]))
```

接下来，我们需要修改 show-guests 函数，让它去调用 db/read-guests：

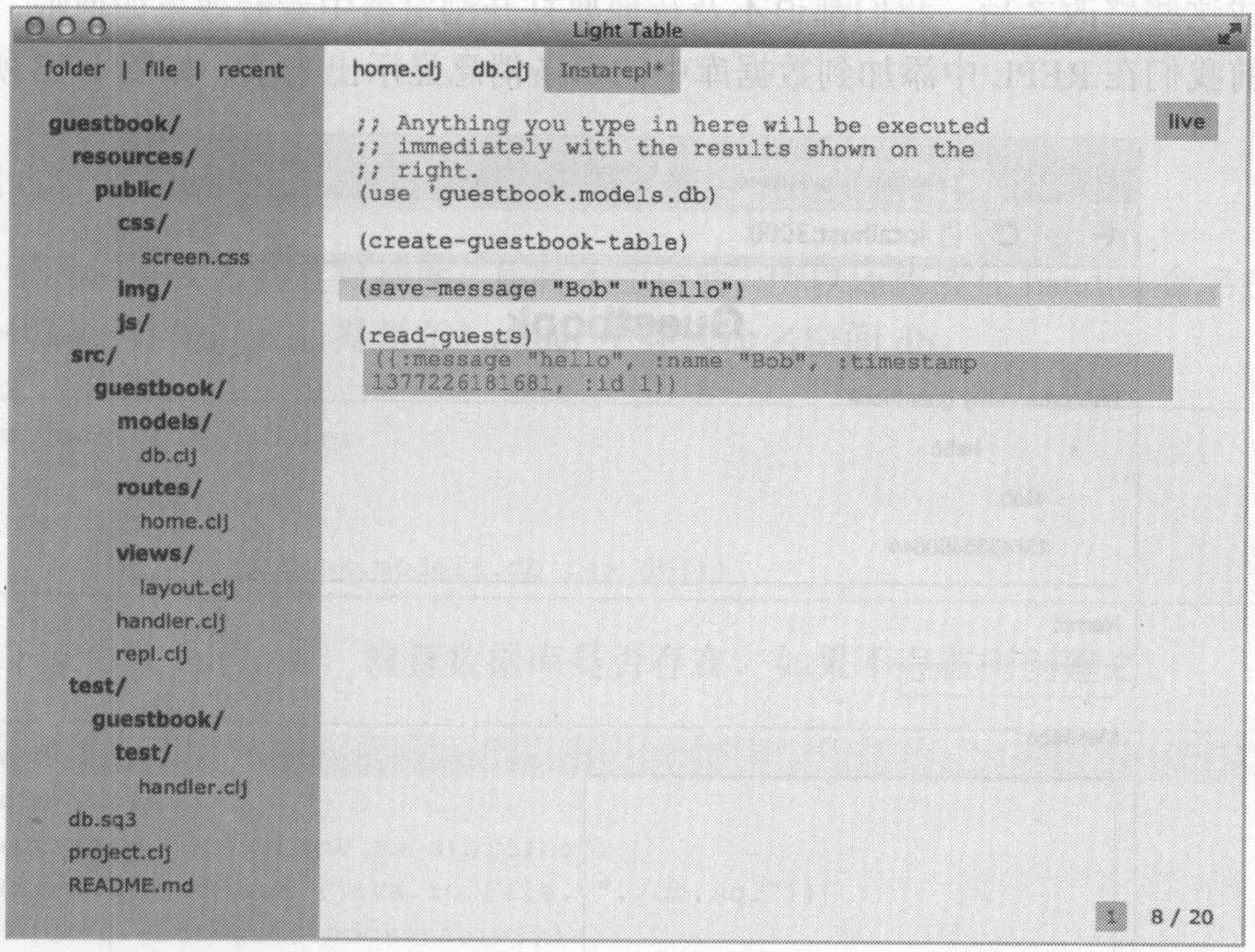

图 1-11 测试保存功能

```
(defn show-guests []
 [:ul.guests
  (for [{:keys [message name timestamp]} (db/read-guests)]
   [:li
    [:blockquote message]
    [:p "-" [:cite name]]
    [:time timestamp]])])
```

最后，我们还得修改 save-message 函数，让它调用 db/save-message，而不是简单地把参数打印出来：

guestbook/src/guestbook/routes/home.clj

```
(defn save-message [name message]
 (cond
  (empty? name)
  (home name message "Some dummy forgot to leave a name")
  (empty? message)
  (home name message "Don't you have something to say?")
  :else
  (do
   (db/save-message name message)
   (home))))
```

完成这些修改之后，我们都迫不及待地要打开浏览器中看看效果如何啦。不出所料，先前我们在 REPL 中添加到数据库中的那条消息显示出来了，如图 1-12 所示。

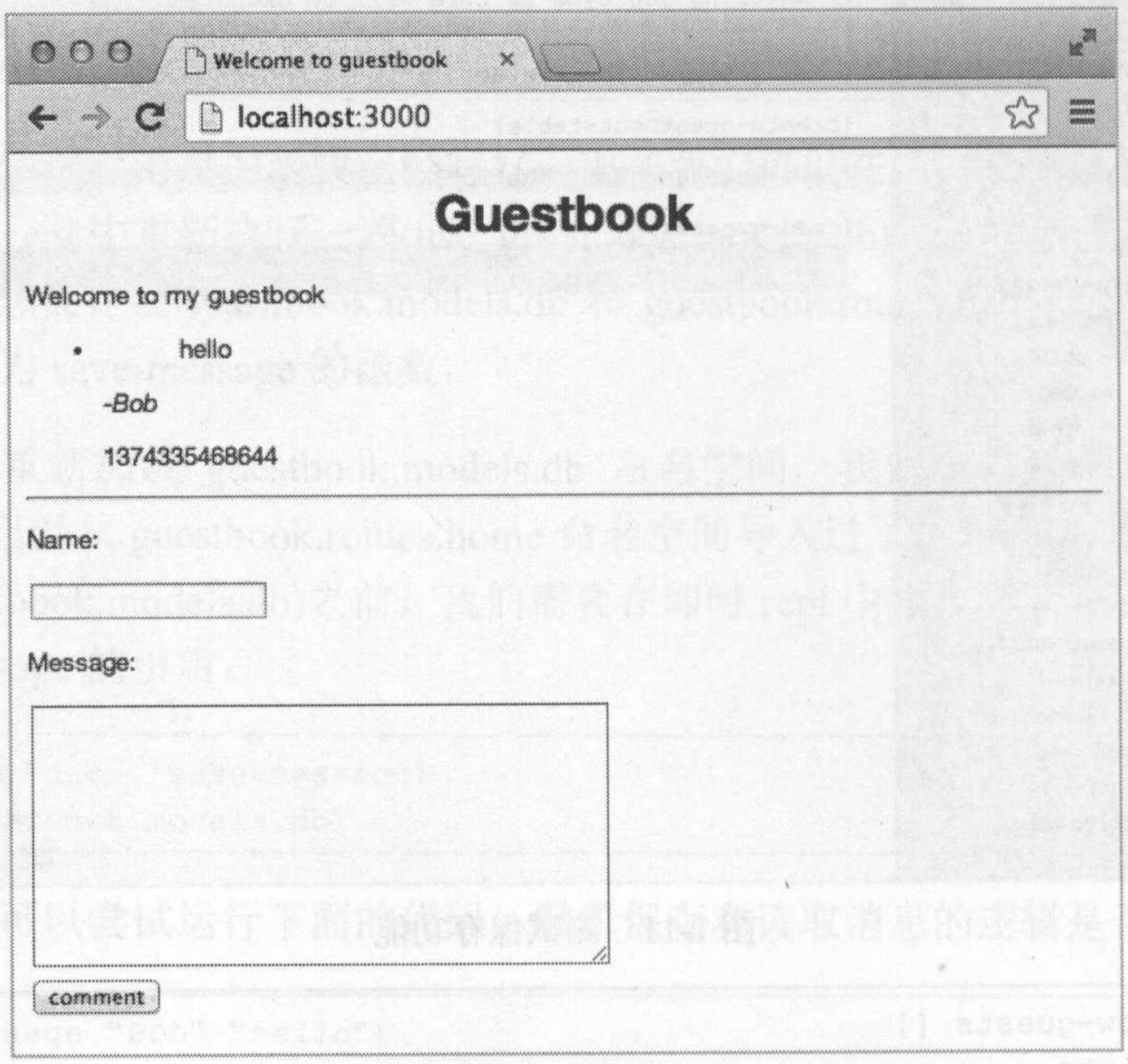

图 1-12　真正的留言

我们还可以试着多录入几条消息，以确认留言簿的功能确实符合预期。

你也许注意到了，页面上消息的显示是存在缺陷的。时间只是简单的显示为毫秒数。这实在是太不友好了，所以，让我们添加一个改善其格式的函数吧。

为此，我们将会创建一个 Java 的 SimpleDateFormat 对象，用来对时间戳进行格式化。

guestbook/src/guestbook/routes/home.clj

```
(defn format-time [timestamp]
  (-> "dd/MM/yyyy"
      (java.text.SimpleDateFormat.)
      (.format timestamp)))
(defn show-guests []
  [:ul.guests
    (for [{:keys [message name timestamp]} (db/read-guests)]
     [:li
        [:blockquote message]
        [:p "-" [:cite name]]
        [:time (format-time timestamp)]])])
```

收尾

我们的留言簿应用已接近完成，还剩下最后一个问题。

由于我们需要先创建数据库，其后才能访问，所以还需要往 handler 命名空间中添加一些代码。首先，我们需要在 handler 中导入命名空间 db。

```
(ns guestbook.handler
 ...
 (:require ...
           [guestbook.models.db :as db]))
```

接下来修改 init 函数，检查数据库是否存在，如果不存在则创建之。

guestbook/src/guestbook/handler.clj

```
(defn init []
  (println "guestbook is starting")
  (if-not (.exists (java.io.File. "./db.sq3"))
    (db/create-guestbook-table)))
```

由于应用加载时会调用 init 函数，因此就能确保数据库在真正开始运行之前便已经准备妥当了。

你学到了什么

通过前面这个例子，我们体验了如何使用 Clojure 来开发 Web 应用。你也许已经注意到了，你只是编写了极少的代码，就得到了一个可用的程序。而且，你几乎没有编写任何样板代码。

阅读至此，你对程序结构、主要组件，以及如何将它们组合到一起应该相当熟悉了。

回顾一下，我们的应用包含了以下几个命名空间。

命名空间 guestbook.handler 的职责是启动服务，并创建一个 handler，负责把来自客户端的请求传递给应用。

然后是命名空间 guestbook.routes.home。我们在这里建立了留言功能的工作流程，同时大部分应用逻辑也都位于此处。如果需要添加更多的工作流，你需要在

guestbook.routes 下创建新的命名空间。例如，你可能会创建 guestbook.routes.auth 命名空间，用来处理用户注册和认证。

通常，routes 文件夹下的每个命名空间都封装着应用中一个自包含的工作流程。所有与之相关的代码都位于同一个地方，并且与其他的路由保持独立。此处工作流表示的可能是用户认证，也可能是内容编辑，或是事务管理等。

命名空间 guestbook.views.layout 负责管理应用的界面布局。我们会在这里放置一些代码，用来生成页面的公共元素，以及控制页面的结构。一般来说，布局方面需要考虑的内容包括：组织静态资源，比如页面需要的 CSS 和 JavaScript 文件；设置公共元素，比如页眉和页脚等。

最后，还有命名空间 guestbook.models.db，它负责整个应用的数据模型。联系例子中定义的数据表，它描述了数据的类型，以及哪些用户的数据需要持久化。

当我们着手构建更大规模的应用时，这些东西应该牢记于胸。一个结构良好的 Clojure 应用会易于理解，也方便维护。对于有些编程语言，当代码规模较大时，你得费尽心思才能理清其复杂的层次结构。而在 Clojure 应用的整个生命周期中，你都不会有类似的烦恼，这真是太美妙了。

我们使用了 Light Table 来开发留言簿应用。虽然它相当易用，但仍需更多打磨，还缺乏一些其他集成开发环境（IDE，Integrated Development Environments）提供的有用特性。这些特性包括代码完成、结构化的代码编辑，以及集成的依赖管理。

所以，我建议你花些时间去尝试一下那些更为成熟的开发环境，例如 Eclipse[①]或者 Emacs[②]。本书的剩余部分假定以 Eclipse 作为我们的开发环境，不过，无论你选用的是哪种编辑器，都没有任何问题。如需了解其他可选的 IDE，不妨参考“附录 1 选择 IDE”。

你会发现，在开发应用的过程中，我们大量使用了 REPL。因此，对于 Clojure 开发环境而言，是否集成了 REPL 可谓有着天壤之别。能在 REPL 中执行代码，就意味着你能获得更快的反馈周期，从而显著地提升生产力。

在本章中，我们演示了如何设置开发环境，以及如何搭建一个典型的 Clojure Web 应用。下一章，我们将关注那些构成 Clojure Web 栈的核心库。你将会了解到以下内容：请求响应的生命周期、定义路由、会话管理，以及利用中间件强化核心处理请求功能。

① http://www.eclipse.org/
② http://www.gnu.org/software/emacs/

第2章

Clojure 的 Web 技术栈

在上一章，我们直接构建了一个简单的应用。通过它，我们对工程的结构有了初步印象，同时也熟悉了开发环境。现在，我们将节奏放缓，先往后退一步，了解一下所有这些组件的运作细节。

Clojure 社区崇尚简单和灵活，而不是循规蹈矩或是一成不变。实际上，Web 栈中的所有组件，都有为数众多的替代品。你可以根据自己的风格，以及你开发的应用类型做出选择。本书中，我们把重点放在流行的 Ring/Compojure 栈，现实当中，许多案例都是用它创建的。

前面的章节中我们介绍了一个简单的应用，用户可以留言并且能够看到其他用户的留言。我们介绍了工程的目录结构和主要文件，以及它们的用途。然而，我们还没有真正关注这些文件中的代码。在本章中，你将学习一些必要的背景知识，以便能够充分地理解我们的这个留言板应用。

由于 Clojure Web 栈是建立在 Java HTTP Servlet API[①]之上的，所以可以将应用部署到任意的 servlet 容器中，比如 Jetty[②]、GlassFish[③]或者是 Tomcat[④]。

你可以选择让 Clojure 应用独立运行，也可以将它和其他 Java 应用一块儿部署在一个应用服务器上。

---

① http://www.oracle.com/technetwork/java/index-jsp-135475.html
② http://www.eclipse.org/jetty/
③ https://glassfish.java.net/
④ http://tomcat.apache.org/

由于众多云服务都支持 Java 虚拟机，你也可以考虑把应用部署到云端。这些服务包括亚马逊的 AWS[①]、谷歌的 App Engine[②]，此外还有 Heroku[③]和 Jelastic[④]等。

servlets 用于处理任意的网络请求/相应交换，这意味着 HTTP servlets 会按照 RFC 描述的有关内容处理 HTTP 交换。servlet 容器调用 servlet 类的接口并传入对应的数据作为请求，并且 servlet 返回用于 servlet 容器的数据作为响应。这套 API 提供了诸多 Web 开发中需要用到的核心功能，比如 cookies、会话，以及 URL 重写。然而，servlets 是专为 Java 语言设计的，如果直接在 Clojure 中使用，未免有些别扭。

不像许多其他平台（比如 Rails 或 Django），Clojure 的 Web 栈并没有提供那种自以为完备的整体框架。相反，你可以把各种库糅合在一起，来构建你自己的应用。本书中，我们仅专注几个常用的 Web 开发库。

作为起点，让我们先了解一下 Ring 和 Compojure 这两个提供了原生 Clojure Servlet API 的库吧。Ring 封装了 Java 的 servlet API，而 Compojure 则用来把请求处理函数映射到指定的 URL。应用本身则位于栈顶，使用这些库来与客户端交互，以及管理应用的状态。

2.1　使用 Ring 来路由请求

Ring 的目标是把 HTTP 的细节抽象为简单且模块化的 API，可以用来构建类型广泛的应用。如果你曾经使用 Python 或是 Ruby 开发过 Web 应用的话，那么你会发现 Ring 与 Python 的 WSGI[⑤]和 Ruby 的 Rake[⑥]非常类似。

对于构建 Web 应用，Ring 已经成为了事实上的标准，因此诞生了很多周边的工具和中间件。尽管在大多数情况下你都无需直接与 Ring 打交道，但高屋建瓴地了解一下其设计，将对后续的开发和排错有颇多益处。

基于 Ring 的应用都包含以下这四个基本组件：处理器（handler）、请求（request）、响应（response）和中间件（middleware）。来分别了解它们一下吧。

---

① http://aws.amazon.com/
② https://developers.google.com/appengine/
③ https://www.heroku.com/
④ http://jelastic.com/
⑤ http://wsgi.readthedocs.org/en/latest/
⑥ http://rack.github.io/

请求处理

Ring 使用标准的 Clojuremap 来表示客户端请求以及服务端响应。而所谓 handler，不过是一组用于处理客户端请求的函数罢了。这些函数的参数是请求 map，返回值则是响应 map。下面是一个非常简单的 Ring handler：

```
(defn handler [request-map]
  {:status 200

   :headers {"Content-Type" "text/html"}
   :body (str "<html><body> your IP is: "
              (:remote-addr request-map)
              "</body></html>")})
```

如你所见，它的参数是一个表示 HTTP 请求的 map，返回了一个表示 HTTP 响应的 map。至于说如何将 HTTP servlet 请求对象转换为 map，以及如何将 map 转换为响应对象，那就是 Ring 操心的问题了。

前面的这个handler只是简单地生成了一段内容为客户端IP地址的HTML字符串，并将响应的状态码置为 200。由于类似这种操作实在是太常见了，于是 Ring 就提供了一个辅助函数用来生成这样的响应：

```
(defn handler [request-map]
  (response

    (str "<html><body> your IP is: "
         (:remote-addr request-map)
         "</body></html>")))
```

如果想要创建自己的响应，你只用编写一个函数，处理传入的请求 map，并返回用于表示你自己响应的 map 即可。下面让我们来了解一下这两个 map 的格式吧。

请求 map 和响应 map

请求 map 和响应 map 都包含了诸如服务端口、URI、对端地址、负载类型以及实际的负载数据。这些 map 的键名源自于 servlet API 和官方的 HTTP RFC 标准文档[①]。

① http://www.w3.org/Protocols/rfc2616/rfc2616.html

请求 map 的内容

请求 map 中定义了下列的标准键。注意，此处列出的键，并不一定会出现在所有的请求中，比如:ssl-client-cert。

- :server-port——用于处理该请求的服务端口。
- :server-name——服务器的 IP 地址或是主机名。
- :remote-addr——客户端的 IP 地址。
- :query-string——请求的查询字符串。
- :scheme——协议的类型，可以是 HTTP 或者 HTTPS。
- :request-method——请求的方法，比如：get、:head、:options、:put、:post 或:delete。
- :request-string——请求的查询字符串。
- :content-type——请求消息体的 MIME 类型。
- :content-length——请求消息体的字节数。
- :character-encoding——请求采用的字符编码名称。
- :headers——包含了请求头部的 map。
- :body——可用于读取请求消息体的输入流。
- :context——当应用没有作为根来部署时，其所处的上下文。
- :uri——服务端的 URI 全路径，包含了:context（如果存在）的部分。
- :ssl-client-cert——客户端的 SSL 证书。

除了上述由 Ring 规定的标准键之外，请求 map 中还有可能会出现由中间件函数添加的其他一些特定于应用的键。怎么才能做到？别着急，本章后面会讨论这个话题。

响应 map 的内容

响应 map 仅包含三个键，就足以描述 HTTP 响应了：

- :status——响应的 HTTP 状态码。
- :headers——返回给客户端的 HTTP 头部。
- :body——响应的消息体。

status 是一个数字，表示 HTTP RFC 标准中定义的一个状态码，规定其最小值为 100。

headers 是一个 map，包含所有表示 HTTP 头部的键值对。头部可以是字符串，也

可以是字符串的序列，在这种情况下，序列中的每个字符串都会作为单独的键和值来发送。

最后，响应消息体可以是一个字符串、一个序列、一个文件或者是一个输入流。此外，消息体还应该与响应的状态码对应。

当响应的消息体是一个字符串时，它会被原样发送给客户端。而如果它是一个序列的话，那么发送给客户端的将是每一个元素的字符串表达。最后，如果响应是一个文件，或者是一个输入流，那么服务器会将其中的内容发送给客户端。

通过中间件扩充功能

所谓中间件，就是一些用来封装处理器（handler）的函数，这些函数能够更改处理请求的方式。中间件函数通常被用于扩展 Ring 的基本功能，以满足应用的实际需要。

中间件本身就是一个函数，它接受一个现有的 handler 和一些其他的可选参数，并返回一个新的 handler，只不过这个新的 handler 将具有一些新的行为特征。下例就是这样的一个函数：

```
(defn handler [request]
  (response
    (str "<html><body> your IP is: "
         (:remote-addr request)
         "</body></html>")))

(defn wrap-nocache [handler]
  (fn [request]
    (let [response (handler request)]
      (assoc-in response [:headers "Pragma"] "no-cache"))))

(def app (wrap-nocache handler))
```

以上例子里封装了一个函数，它接受一个 handler，并返回一个 handler 形式的函数。由于这个返回函数封装在局部，也就可以在处理内部引用 handler。当函数被调用，它就将请求参数传递给 handler 并在回应的 map 里添加 Pragma:no-cache。

这种封装处理称为闭包（closure），因为它隐蔽了 handler 函数的参数内容，使之易于处理返回。

面对程序中的任何具体问题，我们都可以用刚才这种手法创建小函数（small functions）来解决。再将他们通过各种组合，使应用程序可以轻松应对任何复杂的真实环境。

适配器是什么

适配器位于 handler 和 HTTP 框架协议之间，它们组织并提供一些必要的内容，比如端口映射、解析 HTTP 请求，还能通过 handler 返回的 map 构造 HTTP 响应。不过，你并不太需要直接和适配器打交道，我们就此不提了。

2.2　定义 Compojure 路由

Compojure 是构建在 Ring 之上的路由库，它提供的方式非常简洁，用来关联处理 URL 和 HTTP 方法。Compojure 路由基本上是这样子的：

```
(GET "/:id" [id] (str "<p>the id is: " id "</p>" ))
```

其路由函数名与 HTTP 方法名直接对应，比如 GET、POST、PUT、DELETE 和 HEAD。还有一个称为 ANY 的路由会响应客户端任何方法。URI 是包含冒号的键名，对应的那些值可以用作路由参数，Rails[①]和 Sinatra[②]就是使用类似的处理机制，而 Compojure 正是受到这种特性的启发。上面的 Ring 回应描述中会自动包含路由回应。

其实在我们的实际应用中，可能会存在多条路由，Compojure 提供了路由功能，能从多条路由中创建一个 Ring 处理。假设我们有/foo 路由和/:id 项，那么我们可以使用单条处理进行如下合并：

```
(defn foo-handler []
  "foo called")

(defn bar-handler [id]
  (str "bar called, id is: " id))

(def handler
  (routes
    (GET "/foo"[](foo-handler))
     (GET "/bar/:id foo" [id] (bar-handler id))))
```

① http://rubyonrails.org/
② http://www.sinatrarb.com/

定义路由是一种很常见的操作，Compojure 还提供了 defroutes 宏，通过给定的路由生成一个 Ring 处理程序：

```
(defroutes handler
  (GET "/foo" []      (foo-handler))
  (GET "/bar/:id" [id] (bar-handler id)))
```

使用 Compojure 路由，可以非常方便地将网站的每个 URL 映射到功能代码，并且提供 Web 应用的大部分核心功能。我们可以像前面那样，使用 defroutes 宏把这些路由组织起来。大致上，Compojure 就是这样维护 Ring 处理的。

对基于路径共享的程序，Compojure 也提供强大的过滤机制处理常见路由。假设我们现有多条路由来响应特定用户：

```
(defn display-profile [id]
  ;;TODO: display user profile
  )
(defn display-settings [id]
  ;;TODO: display user account settings
  )
(defn change-password [id]
  ;;TODO: display the page for setting a new password
  )
(defroutes user-routes
  (GET "/user/:id/profile" [id] (display-profile id))
  (GET "/user/:id/settings" [id] (display-settings id))
  (GET "/user/:id/change-password" [id] (change-password-page id))
```

现在每条路由前段都是/user/:id，必然会有很多重复代码。我们可以使用 context 宏，来解析路由的相同部分。

```
(def user-routes
     (context "/user/:id" [id]
       (GET "/profile" [] (display-profile id))
       (GET "/settings" [] (display-settings id))
       (GET "/change-password" [] (change-password-page id))))
```

这段代码中，路由定义了与/user/:id 有关的内容，和前一个版本功能完全一样，都能使用 id 参数。context 宏正是通过闭包来实现的。输出 handler 封装了通用参数，它们就可以在内部定义。

访问请求参数

有些路由，需要我们使用请求的 map 保存请求参数。我们通过以下这种方式声明 map，并作为路由的第二参数：

```
(GET "/foo" request (interpose ", " (keys request)))
```

此路由提取请求 map 的所有键名，并罗列出来，其输出如下。

```
:ssl-client-cert, :remote-addr, :scheme, :query-params, :session, :form-params,
:multipart-params, :request-method, :query-string, :route-params, :content-type,
:cookies, :uri, :server-name, :params, :headers, :content-length, :server-port,
:character-encoding, :body, :flash
```

Compojure 同样提供一些实用功能来处理请求 map，包括格式化参数之类。例如，在留言簿程序中（第 1 章“起步”，第 1 页），我们看到如下路由定义：

```
(POST "/" [name message] (save-message name message))
```

这个路由从请求参数中提取了:name 和:message 两个键，然后将它们绑定给同名变量。就像其他的声明变量一样，现在，我们在路由作用范围内就可以使用了。

常规的 Clojure 解构也可用于路由内部，假设给定一个包含如下参数的请求 map：

```
{:params {"name" "some value"}}
```

我们可以使用这种方式从参数中提取“name”关键字：

```
(GET "/:foo" {{value "name"} :params}
  (str "The value of name is " value))
```

此外，Compojure 还提供解构形参子集，并用剩余部分创建一个 map：

```
[x y & z]
x -> "foo"
y -> "bar"
z -> {:v "baz", :w "qux"}
```

以上代码中，参数 x 和 y 都绑定到变量，v 和 w 被重命名为一个名为 z 的 map。

此外，如果我们需要完整的请求参数，我们还可以进行如下处理：

```
(GET "/" [x y :as r] (str x y r))
```

这里，我们将形参绑定给 x、y，还有完整的请求 map 绑定给变量 r。Ring 和 Compojure 装备上函数式这种强劲的武器，我们就能轻易创建页面，并为站点提供路由。但是，完善的应用还需要许多其他的特性，比如页面缓存、会话管理、输入验证，面对这些任务，我们使用最棒的适配器库来逐个击破。

2.3 应用架构

典型的 Compojure 开发 Web 程序方式可能不同于你之前使用的方式。多数框架偏好使用模型-视图-控制器（MVC，model-view-controller）模式使用逻辑分离思想将视图、控制、模式严格分开。这里，Compojure 并没有明确分离视图和控制。

相反，我们为程序中每个路由创建了独立的 handler，这些 handler 用于处理来自客户端的 HTTP 请求，Compojure 正是以这种思路来分派任务的。handler 驱动模型负责处理域逻辑。这种方法提供了一个彻底的域逻辑分离模式，并不牵涉应用程序的表示层，也没有任何不必要的联系。

尽管如此，Clojure 的 Web 栈设计得还是比较灵活，它甚至允许你以任何喜好的方式来组织，如果你非要在程序中使用传统 MVC 风格，也不会有什么麻烦。

仅通过几个逻辑部件就能一览典型应用（这是指我们前面做的那个留言簿程序的结构）。那我们再看看别的一些特性，多数应用被拆分为如下几个方面。

- handler——此命名空间负责处理请求、响应。
- routes——路由涵盖我们程序的核心内容，譬如维护读取页面和处理客户端请求的逻辑关系。
- model——此命名空间保留给数据模型和持久化层。
- views——此命名空间包含通用逻辑以构成应用层。

程序的 handler

handler 是功能入口，它通常用于定义 handler 命名空间。它负责将程序的所有路由汇聚起来，并且定义所有的处理过程，用于封装必要的中间件。

handler 命名空间也为程序定义一些基础路由，但不用于任何特定的工作流。我们留言簿程序中的那个 handler，有两条路由：一条用于处理静态资源；还有一条用于捕获其他所有路由都未定义的 URI 请求。

```
(defroutes app-routes
  (route/resources "/")
  (route/not-found "Not Found"))
```

路由里具体的工作流，比如在留言簿里发布和浏览消息的路由处理，都组织在与它们功能相关的特定命名空间里。每一条都供 routes 命名空间访问。

handler 命名空间也提供 init 和 destroy 方法，它们在程序起停时被调用。任何需要在始末阶段调用的代码，都要分别放在这两个函数里面执行。

举个例子说明吧，我们在留言簿程序里就用上了，init 函数用来检查数据库连接是否可用。

```
(defn init []
  (println "guestbook is starting")
  (if-not (.exists (java.io.File. "./db.sq3"))
    (db/create-guestbook-table)))
```

接下来，我们定义入口点，在调用 app 函数时，程序将开始处理所有路由请求。

```
(def app (handler/site (routes home-routes app-routes)))
```

这段代码，compojure.handler/site 函数用于生成 Ring handler，用中间件支撑一个典型网站。

site 函数仅仅创建一个 handler，并将其封装进一些通用中间件，来支持通用网站。中间件由如下封装器构成。

- wrap-session。
- wrap-flash。
- wrap-cookies。
- wrap-multipart-params。
- wrap-params。
- wrap-nested-params。
- wrap-keyword-params。

在 project.clj 里，程序的 handler、init 函数、destroy 函数，都绑定在:ring 键下面，具体参见我们的留言簿程序（“第 1 章起步”）。

```
:ring {:handler guestbook.handler/app
       :init    guestbook.handler/init
       :destroy guestbook.handler/destroy}
```

以上描述用于引导程序核心部分。接下来，我们一起看看怎样添加一些别的路由，来满足应用程序的具体功能。

路由请求

此前我们讨论过，程序路由表现为 URI，由客户端请求，由服务端执行。客户端请求的 URI 由路由程序对应的处理函数做相应回应。

现实当中没有哪个应用只有一条路由。比如，在我们的留言簿程序中，有两个独立路由，各自执行不同的操作：

guestbook/src/guestbook/routes/home.clj

```
(defroutes home-routes
 (GET "/" [] (home))
 (POST "/" [name message] (save-message name message)))
```

第一条路由被绑定于/，用于从数据库检索消息，并用此消息创建一张表单，最终呈现整幅页面给客户端。

第二条路由会处理用户输入。如果输入验证通过，接下来这条消息就会被存入数据库；否则，页面将呈现错误描述。

其实这两条路由功能有交集：存储和显示用户信息，它们也算是同一工作流的两个部分。

当你发现程序的工作流有明确所属，那么可以将此工作流的逻辑关系合并，放在一起处理。程序中的 routes 包之下的命名空间正是为这种特殊工作流预留的。

由于我们的留言簿应用很小。除了在 guestbook.routes.home 命名空间里有几个辅助函数，定义一套路由就够用了。

当程序包含多个页面，为便于维护代码，我们会创建额外的命名空间。接下来我

们用 Compojure 提供的 routes 宏，在每个独立的命名空间下创建独立的路由，并将处理放在 handler 命名空间。

routes 宏可以将多个路由合并，最终创建 handler。有一点要注意，路由之间存在覆盖关系。由于我们的 app-routes 调用了(route/not-found "Not Found")，务必把它置为最后一条，否则在 not-found 路由后面的所有路由将被覆盖。

应用模型

稍稍复杂一些的应用，都需要建立在某种模型之上。模型用于描述应用程序如何存储数据、单个数据元素之间的内在关系。我们的留言簿程序模型由用户表和消息表构成。

处理模型和持久层的所有命名空间，惯例上属于 models 包。我们在下一章会用大篇幅重点讲述。

应用视图

views 包用于为页面提供可视布局和其他的通用控件，其下有预设的 layout 命名空间。这个命名空间为我们包含了 common 布局声明，用于生成基础页面模板。

common 布局用于填充页面头、填写标题标签、打包资源（如 CSS）及添加负载内容。由于内容使用 html5 宏封装，common 布局被调用之后，将自动创建 HTML 文本串，这个处理直接将结果反馈给客户端。

这种方式常用于创建通用布局，以及提供基本页面结构，也使用它定义个别页面。亦可创建通用页面元素，比如页眉、页脚、菜单，并会得到统一维护。我们每次创建的页面，都需要使用定义的布局简单将内容包裹起来。

定义页面

创建路由的同时也就定义了页面，通过接受请求参数来生成各种特殊的响应，比如用来返回 HTML 元素，执行服务端操作，重定向到另一个页面；或者返回特殊类型的数据，比如数据交换格式（JSON，JavaScript Object Notation）字符串或文件。

通常，一张页面由多条路由组成。其中有一条接受 GET 请求，并返回 HTML 供浏览器渲染的路由。还有其他情况，比如在客户端用户与页面交互时，生成并提交了表单，这时会有其他路由来处理此请求。

无论我们选择如何处理，都能创建页面，Compojure 并不关心我们使用的具体方法，这恰好为选择模板库留有余地。可选的方案不少，这里介绍几个流行的库：Hiccup[①]、Enlive[②]、Selmer[③]、Stencil[④]。

Hiccup 能使用原生 Clojure 数据结构，通过它定义表情并生成相适应的 HTML；Enlive 反其道而行，使用纯 HTML 定义页面而不用特殊处理标签。适配器将特定模型和域变换为 HTML 模板。

与 Hiccup 和 Enlive 不一样，Stencil 和 Selmer 都是基于外部模板系统，而不是基于 Clojure。Stencil 是实现了 Mustache（这是个流行的无逻辑模板系统），Selmer 是模仿 Django 模板系统在 Python 上的实现。

本书重点关注并使用 Hiccup，因为它不需要额外学习任何语法，直接使用 Clojure 函数即可。此外，我们在后面还会学习用 Selmer 模板来取代 Hiccup 创建的应用。

别的选择彻底没有考虑使用服务端模板，你需要在客户端处理模板来接管这些工作，挑个流行的 JavaScript 库，并使用 Ajax 与服务通讯。当然，这样也能胜任。好处是这可以让客户端服务端的界限明确、清晰，有助于扩充其他形式的客户端，比如移动应用接口。在编写单页应用[⑤]时，这还是通行手段。

无论你喜欢何种模板策略，最佳实践都不会去聚合域逻辑和视图。通过合理构架的程序，是可以轻松替换模板引擎的。

Hiccup 处理模板化页面

现在开始介绍一些 Hiccup 使用基础，以及通过它如何生成适当的页面元素。

刚才提到，用原生 Clojure 就能编写 Hiccup 模板，所以你就不需要去学习**特定领域语言**（DLS，domain-specific language）就能驾驭它。

---

① https://github.com/weavejester/hiccup
② https://github.com/cgrand/enlive
③ https://github.com/yogthos/Selmer
④ https://github.com/davidsantiago/stencil
⑤ http://en.wikipedia.org/wiki/Single-page_application

Hiccup 用 Clojure vector（向量表）表示 HTML 元素，其属性使用 map 描述，这种结构表达方式与生成的 HTML 标签在结构上比较吻合，示例如下。

```
[:tag-name {:attribute-key "attribute value"} tag body]
<tag-name attribute-key="attribute value">tag body</tag-name>
```

如果我们想要创建一个包含图片的 div 标签，可以创建一个 vector，第一个元素为:div 关键字，紧随其后是一个 map（包含 div ID 和 div 的 class）。余下部分是以 vector 表示图片的内容构成。

```
[:div {:id "hello", :class "content"} [:p "Hello world!"]]
```

我们使用 hiccup.core/html 宏将 vector 转换为 HTML 文本：

```
(html [:div {:id "hello", :class "content"} [:p "Hello world!"]])
<div id="hello" class="content"><p>Hello world!</p></div>
```

由于 Hiccup 允许你通过 map 设置元素属性，如有必要，你还可以使用元素内联样式。尽管如此，你还是应该抵御这种诱惑，使用 CSS 样式化元素取代之，这可以确保结构和描述分离。

由于对元素设置 ID 和设置 class 是常用操作，Hiccup 还提供便捷的 CSS 样式化处理。我们可以如下简化编写我们的 div，取代之前的代码：

```
[:div#hello.content [:p "Hello world!"]]
```

Hiccup 同样提供一些辅助函数，用来定义常用元素，比如表单、链接、图像。所有这些函数输出的 vector，由 Hiccup 预先定义的格式描述。

当一个函数在使用中并不能满足需求时，你当然可以写下元素的文本描述，还可以调整输出来满足需要。描述 HTML 元素的函数可以配置，其第一个参数可以接受可选属性的 map。我们再了解一些常用的 Hiccup 辅助函数，来改善使用体验。

首先，我们来看看怎么用 link-to 辅助函数创建一个标签：

```
(link-to {:align "left"} "http://google.com" "google")
```

这段代码将生成以下 vector：

```
[:a {:align "left", :href #<URI http://google.com>} ("google")]
```

我们已有一个关键字:a作为第一项，紧随其后的map表示属性，以及表示内容的list。还是如此，将 link-to 函数封装在 html 宏里面，我们可以基于此 vector 输出 HTML：

```
(html (link-to {:align "left"} "http://google.com" "google"))
<a align="left" href="http://google.com">google</a>
```

还有一个常用的函数 form-to，用来生成 HTML 表单，我们用此函数实现上一章创建的表单，并将信息提交给服务端。

```
(form-to [:post "/"]
         [:p "Name:" (text-field "name")]
         [:p "Message:" (text-area {:rows 10 :cols 40} "message")]
         (submit-button "comment"))
```

这个辅助函数接受一个 vector，第一个元素是 HTTP 请求类型的关键字，第二个元素是 URL 字符串。余下参数也为 vector，通过求值可以表示为 HTML 元素。当调用 html 宏后，前面的代码会被转化为以下 HTML：

```
<form action="/" method="POST">
  <p>Name:<input id="name" name="name" type="text" /></p>
  <p>Message:<textarea cols="40" id="message" name="message" rows="10">
  </textarea></p><input type="submit" value="comment" />
</form>
```

还有一个实用的辅助宏 defhtml。我们在定义一个函数同时，通过参数内容悄悄生成 HTML。这意味着在构造页面时，我们不需要用 html 宏作用每一个独立元素。

```
(defhtml page [& body]
  [:html
    [:head
     [:title "Welcome"]]
     [:body body]])
```

同样，在 hiccup.page 命名空间里，Hiccup 提供若干生成特定 HTML 变体的宏，比如 HTML4、HTML5 和 XHTML。看，我们在留言簿程序里使用的就是 HTML5 宏。

```
(defn common [& body]
  (html5
   [:head
    [:title "Welcome to guestbook"]
    (include-css "/css/screen.css")]
   [:body body]))
```

添加资源

现实中，大型网站的页面必然涉及加载 JavaScript 和 CSS。在 hiccup.page 命名空间里，Hiccup 提供几个实用函数来达到这个目的。你可以使用 include-css 去引用任何 CSS 文件，include-js 来加载 JavaScript 资源。这里有个在常用布局中包含 CSS 和 JavaScript 资源的例子：

```
(defn common [& content]
  (html5
   [:head
    [:title "My App"]
    (include-css "/css/mobile.css"
                 "/css/screen.css")
    (include-js "//code.jquery.com/jquery-1.10.1.min.js"
                "/js/uielements.js")]
   [:body content]))
```

如你所见，include-css 和 include-js 都能接受多个字符串，每个参数指定一个 URI 资源。它们的输出必然是一个 Hiccupvector，最终会被转换为 HTML。

```
;;output of include-css
([:link
  {:type "text/css", :href #<URI /css/reset.css>, :rel "stylesheet"}]
 [:link
  {:type "text/css", :href #<URI /css/screen.css>, :rel "stylesheet"}])

;;output of include-js
([:script
  {:type "text/javascript",
   :src
   #<URI //code.jquery.com/jquery-1.10.1.min.js>}]
 [:script {:type "text/javascript", :src #<URI /js/uielements.js>}])
```

同样，在 hiccup.element 命名空间，Hiccup 提供一个名为 image 的辅助函数去加载图片：

```
(image "/img/test.jpg")
[:img {:src #<URI /img/test.jpg>}]
(image "/img/test.jpg" "alt text")
[:img {:src #<URI /img/test.jpg>, :alt "alt text"}]
```

Hiccup API 一览

你已经见识了一些常用的函数，其实还有一些更有用的。大多数辅助函数可以在 element 和 form 命名空间里找到。这些函数用于定义元素，比如图像、链接、脚本标签、复选框、下拉工具栏以及输入栏。

如你所见，Hiccup 提供一套简明 API 去生成 HTML 模板，此外还有字面量 vector 表达式。既然你已经领悟到了 Hiccup 的精髓，那我们回过来对此前的留言簿程序进行更深入的剖析。

回顾留言簿程序

我们现在换个角度去看待那些定义在 home 命名空间的函数。当你试着运行程序，并来回浏览时，顺便查阅页面的 HTML 输出和在代码里的定义。

首先，我们用 show-guests 函数去生成一个无序清单。它遍历数据库的消息，然后为每一个消息创建一个列表项。

```
(defn show-guests []
 [:ul.guests
   (for [{:keys [message name timestamp]} (db/read-guests)]
    [:li
       [:blockquote message]
       [:p "-" [:cite name]]
       [:time (format-time timestamp)]])])
```

这里有个辅助函数，可以用于显示格式化时间戳。此函数使用 java.text.SimpleDateFormat 将日期对象转化为格式化字符串。我们使用流化（->）宏去执行格式化器去格式化文本，接下来使用此方法处理从数据库获取的时间戳。

```
(defn format-time [timestamp]
  (-> "dd/MM/yyyy"
      (java.text.SimpleDateFormat.)
      (.format timestamp)))
```

你可能已经发现目前的 home 函数编写得有点复杂，因为它还有一些用来指导用户提交表单的额外描述。

这里有一点值得一提：错误处理行的代码用于显示错误键值，由控制器填充，最

终交由show-guests函数去呈现内容。

home函数使用layout/common封装内容，为页面生成HTML。

```
(defn home [& [name message error]]
  (layout/common
    [:h1 "Guestbook"]
    [:p "Welcome to my guestbook"]
    [:p error]
    (show-guests)
    [:hr]
    (form-to [:post "/"]
      [:p "Name:" (text-field "name" name)]
      [:p "Message:" (text-area {:rows 10 :cols 40} "message" message)]
      (submit-button "comment"))))
```

如你所见，仅需少许代码，就能使用Hiccup创建页面模板，同时也便于通过关联模板定义生成输出元素。

我们就此完成了路由定义，Compojure路由得以完善。

```
(defroutes home-routes
  (GET "/" [name message error] (home name message error))
  (POST "/" [name message] (save-message name message)))
```

到目前为止，我们已完成创建路由并由此呈现页面，还能处理来自客户端的请求表单。正如我们先前提到的，除了由Ring和Compojure提供的，真实的应用还需要添加一些别的元素。接下来，让我们看看如何为我们的应用添加更多功能。

2.4　Compojure和Ring之后

不少程序库能有效应对各种处理任务，比如会话管理、输入验证、身份认证。你依旧可以随意挑拣适合你的部件。

我们选择lib-noir[①]作为接下来的关注重点，因为应对Web程序的绝大多数任务，它都能胜任。我们之前通过介绍Hiccup的API，学习了它的一些特性及常见功能，同样，我们也来看看lib-noir是如何用的。

首先，为了能启用lib-noir，我们需将其添入项目描述文件project.clj。具体是在依赖项的vector里添加[lib-noir "0.7.6"]。

① https://github.com/noir-clojure/lib-noir

如果你的项目还正运行着，你务必先重启应用，让依赖项生效。接下来，我们再看看如何使用 lib-noir 为应用添加功能。

处理重定向

有些情况下，在执行某些操作之后，我们需要刻意将页面跳转到别的页面。比如，用户在注册页面完成账户注册之后，需要将用户重定向到主页。

既然要实现用户注册，我们就先添加一个注册页吧。第一步，新建一个命名空间，名为 guestbook.routes.auth。与 home 命名空间的处理一样，需要引用其他的命名空间：

```
(ns guestbook.routes.auth
  (:require [compojure.core :refer [defroutes GET POST]]
            [guestbook.views.layout :as layout]
            [hiccup.form :refer
             [form-to label text-field password-field submit-button]]))
```

这个函数用于为我们呈现页面，并会为展示给用户一个表单，用于引导用户输入 ID 和密码。

```
(defn registration-page []
  (layout/common
    (form-to [:post "/register"]
             (label "id" "screen name")
             (text-field "id")
             [:br]
             (label "pass" "password")
             (password-field "pass")
             [:br]
             (label "pass1" "retype password")
             (password-field "pass1")
             [:br]
             (submit-button "create account"))))
```

看得出来，函数内部的表达方式有点累赘，每一个输入需要一个标签，然后还得添加一个换行。好在 Hiccup 使用标准 Clojure 数据结构表述，我们可以提取重复元素，抽象并构造一个辅助函数：

```
(defn control [field name text]
  (list (label name text)
        (field name)
```

```
        [:br]))

(defn registration-page []
  (layout/common
    (form-to [:post "/register"]
      (control text-field :id "screen name")
      (control password-field :pass "Password")
      (control password-field :pass1 "Retype Password")
      (submit-button "Create Account"))))
```

平时，我们会用一个 vector 来直接表述，但这次创建的函数使用 list 函数来包装。这是因为 Hiccup 使用 vector 来表达 HTML 标签，但是标签内容并不能用 vector 来表达。

既然已经创建了新页面，同时也要考虑为其增加一条对应的路由。这里，将路由处理封装到名为 auth-routes 的函数中：

```
(defroutes auth-routes
  (GET "/register" [_] (registration-page)))
```

上面的函数形参 vector 中使用了下划线（_），用在被执行的函数不使用此参数时，这种表达方式是 Clojure 约定俗成的用法。

由于我们已经创建了一条新路由，同样，我们也需要去更新我们的程序处理。我们需要在 handler 命名空间中引用这个新命名空间，同时为我们的程序添加路由，具体如下：

```
(ns guestbook.handler
  ...
  (:require ...
    [guestbook.routes.auth :refer [auth-routes]]))

...

(def app
  (handler/site
    (routes auth-routes home-routes app-routes)))
```

注意，因为路由中使用了(route/not-found "Not Found")，这条路由会覆盖所有定义在此之后的其他路由，新路由应该添加在 app-routes 前段。

如果你已经在 REPL 中运行着站点，那么你需要重启，让新的路由生效。

网站重启之后，则需要导航至 http://localhost:3000/register 确认页面能否正确加载。

如果一切顺利，你现在就可以为注册页面添加处理了。

在成功注册之后，处理会将用户重定向到 home 页。重定向是个简单的 map，包含状态、头、消息体：

```
{:status 302, :headers {"Location" "/"}, :body ""}
```

Ring 在 ring.util.response 命名空间中提供了重定向功能。由于我们已经启用了 lib-noir，使用 noir.response/redirect 取代之。lib-noir 允许使用操作关键字表达重定向状态码。默认是:found，对应的重定向状态码是 302。

我们需要引用这个命名空间才能访问它，将其添加到 auth 命名空间的:require 表中。

```
(ns guestbook.routes.auth
  (:require ...
            [noir.response :refer [redirect]]))
```

现在我们可以在 auth-routes 定义中添加我们的 handler。此刻，我们对输入密码做简单匹配检查判定，成功则重定向到 home 页，否则，我们刷新此页。

```
(defroutes auth-routes
  (GET "/register" [] (registration-page))
  (POST "/register" [id pass pass1]
        (if (= pass pass1)
          (redirect "/")
          (registration-page))))
```

管理会话

在用户与程序交互过程中，我们需要以某种途径去记录用户会话状态。所幸 lib-noir 在 noir.session 命名空间已提供了一套管理会话的方法。将客户端会话表示为一个 map 用于记录，使用如下辅助函数来处理：

- clear! ——清除会话一切内容。
- flash-put ——将一个值储存入检索表。
- flash-get —— 取回一个值并清除之。
- get —— 从会话获取一个值。
- put! —— 将一个值存入会话。
- remove! ——从会话删除一个值。

函数名后缀使用感叹号（!），说明此举会改变会话状态，这种通过在函数名上增加符号来表达操作的表示方式，是 Clojure 约定俗成的。让我们看个例子——实现 login 和 logout 页面，每个动作将对会话做对应更新。

使用 lib-noir 会话的同时，我们会封装 app handler 来访问会话中间件。由于标准处理并不关心会话，也并不在请求之间提供方法去持有状态，所以这种处理是有必要的。

中间件要求我们自己提供储存方式，这样会话状态将会得到持久化处理。可以使用 Redis[①]存于内存或备份至外部存储。

在我们的应用中，我们简单使用 ring.middleware.session.memory/memory-store 来说明。首先在每个中间件和存储处理都要声明引用此命名空间。

```
(ns guestbook.handler
  ...
  (:require ...
    [noir.session :as session]
    [ring.middleware.session.memory
      :refer [memory-store]]))
```

下一步，我们将使用会话中间件封装我们的应用。wrap-noir-session 中间件接受一个包含:store 键的 map 参数。我们绑定此键到 memory-store：

```
(def app
  (->
    (handler/site
      (routes auth-routes
              home-routes
              app-routes))
    (session/wrap-noir-session
      {:store (memory-store)})))
```

现在我们看到的内容涉及创建登录页面并将用户添加到会话。我们打开 auth 命名空间，将如下函数添加入内：

```
(defn login-page []
  (layout/common
    (form-to [:post "/login"]
      (control text-field :id "screen name")
      (control password-field :pass "Password")
      (submit-button "login"))))
```

① http://redis.io/

此函数创建一个包含用户 ID 和密码的登录表单，并使用通用布局封装。当用户点击提交按钮，表单会将一个 HTTP 发送给/login URI。

我们现在更新这个路由定义，为程序创建一个 GET 和 POST 的/login 路由。为使其正常工作，我们同样需要在路由页面引用 noir.session。

```
(ns guestbook.routes.auth
  (:require ...
            [noir.session :as session]))

...

(defroutes auth-routes
  (GET "/register" [] (registration-page))
  (POST "/register" [id pass pass1]
        (if (= pass pass1)
          (redirect "/")
          (registration-page)))
  (GET "/login" [] (login-page))
  (POST "/login" [id pass]
        (session/put! :user id)
        (redirect "/")))
```

GET login 路由简单调用 login-page 函数去显示页面。在重定向到 home 页面之前，POST login 路由使用 noir.session/put!函数和:user 键将用户添加到会话。现在我们将浏览器定位到/login 页面，试试新添加的功能。

对于会话中的那个用户，在我们的 home 函数构造页面的同时，可以调用(session/get :user)来查看，这样就能在更新 home 页面的同时显示用户 ID。此举须先在 home 命名空间声明处放置 noir.session 的包含引用。

```
(ns guestbook.routes.home
  (:require ... [noir.session :as session])
```

guestbook-with-auth/src/guestbook/routes/home.clj

```
(defn home [& [name message error]]
  (layout/common
    [:h1 "Guestbook " (session/get :user)]
    [:p "Welcome to my guestbook"]
    [:p error]

    (show-guests)
```

```
    [:hr]
    (form-to [:post "/"]
             [:p "Name:" (text-field "name" name)]
             [:p "Message:" (text-area {:rows 10 :cols 40} "message" message)]
             (submit-button "comment"))))
```

下一步，我们在创建注销页面时调用noir.session/clear!。当用户单击退出按钮，接下来将会清除此用户在会话中积累的一切信息。

```
(defroutes auth-routes
  (GET "/register" [] (registration-page))
  (POST "/register" [id pass pass1]
        (if (= pass pass1)
          (redirect "/")
          (registration-page)))

  (GET "/login" [] (login-page))
  (POST "/login" [id pass]
        (session/put! :user id)
        (redirect "/"))
  (GET "/logout" []
       (layout/common
         (form-to [:post "/logout"]
           (submit-button "logout"))))
  (POST "/logout" []
        (session/clear!)
        (redirect "/")))
```

切记，session命名空间必须在请求上下文时访问，这意味着不能在路由声明之外使用。

处理输入验证

当创建表单时，我们需要某种途径去检查填写正确与否，并且还需要通知用户关于填写遗漏或项缺失。到目前为止，我们仅简单在参数中填充错误键并显示在页面上。

还是使用类似的办法，我们使用cond实现决策处理：显示有错误描述的登录页面，或者将用户添进会话并重定向页面：

```
(defn login-page [& [error]]
  (layout/common
    (if error [:div.error "Login error: " error])
    (form-to [:post "/login"]
      (control text-field :id "screen name")
      (control password-field :pass "Password")
      (submit-button "login"))))
(defn handle-login [id pass]
  (cond
    (empty? id)
    (login-page "screen name is required")
    (empty? pass)
    (login-page "password is required")
    (and (= "foo" id) (= "bar" pass))
    (do
      (session/put! :user id)
      (redirect "/"))
    :else
    (login-page "authentication failed")))
```

下一步，我们更新 POST /login 路由，使用 handle-login 函数作为 handler 去处理。

```
(POST "/login" [id pass]
  (handle-login id pass))
```

尽管这种方式简单、可用，为了扩充更多规则，很快就会变得乏味。正好 lib-noir 提供了 noir.validation 命名空间，可以使用优雅的方式去处理输入验证。我们在 auth 命名空间引用它，见识一下它如何改善我们的验证处理。

```
(ns guestbook.routes.auth
  (:require ...
            [noir.validation
             :refer [rule errors? has-value? on-error]]))
```

对于使用验证函数，我们一样需要将 handler 封装到 wrap- noir-validation 中间件。这里需要引用 noir.validation：

```
(ns guestbook.handler
  ...
  (:require ...
            [noir.validation
             :refer [wrap-noir-validation]]))
```

guestbook-with-auth/src/guestbook/handler.clj

```
(def app
  (->
    (handler/site
     (routes auth-routes
             home-routes
             app-routes))
     (wrap-base-url)
     (session/wrap-noir-session
       {:store (memory-store)})
     (wrap-noir-validation)))
```

顺便说一声，如果你正运行着 REPL，现在你需要通过重新加载程序来重编译路由。

这里有个 noir.validation/rule 辅助函数，可以取代 cond 来实现决策。每个规则都对内容判定，检查各自是否能通过。最后，函数会调用 noir.validation/errors?去检查规则中是否产生错误。如果有，我们就显示登录页面；否则我们将用户记录到会话，并重定向到 home 页面。

```
(defn handle-login [id pass]
  (rule (has-value? id)
        [:id "screen name is required"])
  (rule (= id "foo")
        [:id "unknown user"])
  (rule (has-value? pass)
        [:pass "password is required"])
  (rule (= pass "bar")
        [:pass "invalid password"])

  (if (errors? :id :pass)
    (login-page)

    (do
      (session/put! :user id)
      (redirect "/"))))
```

我们按如下格式创建规则：

```
(rule validator [:field-name "error message"])
```

验证器可以表达为任何形式，只要最终返回布尔值即可。也可以为每个键设置多重错误，这些错误会被汇集到一个 vector。当验证器返回 false，将生成错误。

例如，我们写下(= id "foo")，id 的值只要不是 foo，就会生成错误。

我们这里为每一个项分别提供一个错误处理。其实可以创建一个辅助函数，用于将它们汇集起来，并统一为展示错误内容做进一步处理。

guestbook-with-auth/src/guestbook/routes/auth.clj

```
(defn format-error [[error]]
  [:p.error error])
```

我们现在更新 control 函数，在调用 on-error 时，传入控制名。这便实现了错误汇聚，对提供的键名使用 format-error 格式化。

guestbook-with-auth/src/guestbook/routes/auth.clj

```
(defn control [field name text]
  (list (on-error name format-error)
        (label name text)
        (field name)
        [:br]))
```

由于我们不再需要将错误定向到 login-page，我们更新对应内容。

guestbook-with-auth/src/guestbook/routes/auth.clj

```
(defn login-page []
  (layout/common
    (form-to [:post "/login"]
      (control text-field :id "screen name")
      (control password-field :pass "Password")
      (submit-button "login"))))
```

总而言之，我们可以在需要验证的任何地方创建规则。每个规则会考察、判定此处是否合法。如果此处验证失败，就会生成错误内容并通过 on-error 辅助函数呈现给用户。

我们之所以可以这样做，是因为验证错误一定是当前的请求带来的。由于调用的这个函数为当前的请求负责处理和展现结果，所以它也应当处理对应的错误。

添加安全机制

Lib-noir 同样提供便捷途径去处理 hash，并使用 noir.util.crypt 验证密码。这个命名空间提供两个名为 encrypt 和 compare 的函数。前者用于密码加密、加盐（salts），后者用于对比明文密码和由前者生成的 hash 字符串。实际上，内部具体使用的是流行的 jBCrypt 库[①]处理的加密。

① http://www.mindrot.org/projects/jBCrypt/

使用 compare 函数去验证看起来是这样：

```
(compare raw encrypted)
```

encrypt 函数允许指定加盐，也生成并提供一个不加盐的版本。

```
(encrypt salt raw)
(encrypt raw)
```

我们之所以对密码加盐，是为了对抗彩虹表 (rainbow-table)[①] 的攻击。彩虹表其实是预先将很多常见密码通过哈希计算生成的字典。此表是通过优化提高哈希查找效率，并且允许攻击者容易通过给定的哈希值来获取密码原文。而加盐操作是为密码追加随机内容再进行哈希，最终生成的哈希便不再容易被破解。

这里，我们同样需要在 auth 命名空间中添加引用：

```
(ns guestbook.routes.auth
  (:require ...
            [noir.util.crypt :as crypt])
```

至此，我们已经将用户状态保存在会话记录中。接下来，我们再看看当用户注册到站点时，如何固化用户详细信息。首先，我们在 db 命名空间下添加几个函数，用于访问数据库：实现一个写操作函数去添加用户，一个读操作函数检索用户。

guestbook-with-auth/src/guestbook/models/db.clj

```
(defn create-user-table []
  (sql/with-connection
    db
    (sql/create-table
      :users
      [:id "varchar(20) PRIMARY KEY"]
      [:pass "varchar(100)"])))

(defn add-user-record [user]
  (sql/with-connection db
    (sql/insert-record :users user)))

(defn get-user [id]
  (sql/with-connection db
    (sql/with-query-results
      res ["select * from users where id = ?" id] (first res))))
```

① http://en.wikipedia.org/wiki/Rainbow_table

完成这些之后，我们需要重新加载 db 命名空间，使得新的函数生效，然后在 REPL 控制台运行(create-user-table)。

我们现在可以切换到 auth 命名空间，开始编写 handle-registration 函数。记住，我们一样也要在 db 命名空间声明引用。

```
(ns guestbook.routes.auth
  (:require ... [guestbook.models.db :as db]))
```

guestbook-with-auth/src/guestbook/routes/auth.clj

```
(defn handle-registration [id pass pass1]
  (rule (= pass pass1)
        [:pass "password was not retyped correctly"])
  (if (errors? :pass)
    (registration-page)
    (do
      (db/add-user-record {:id id :pass (crypt/encrypt pass)})
      (redirect "/login"))))
```

更新 POST /register 路由，这些功能在被调用时将会生效。

```
(POST "/register" [id pass pass1]
      (handle-registration id pass pass1))
```

接下来，当一个用户试图登录时，我们会在登录处理函数中检查其授权。

guestbook-with-auth/src/guestbook/routes/auth.clj

```
(defn handle-login [id pass]
  (let [user (db/get-user id)]
    (rule (has-value? id)
          [:id "screen name is required"])
    (rule (has-value? pass)
          [:pass "password is required"])
    (rule (and user (crypt/compare pass (:pass user)))
          [:pass "invalid password"])
    (if (errors? :id :pass)
      (login-page)
      (do
        (session/put! :user id)
        (redirect "/")))))
```

我们使用 crypt/compare 函数去比对此时提供的密码和其在注册中创建的哈希版本。

指定 MIME 类型

出于一些原因，我们可能会希望明确指定负载内容的类型，比如纯文本、JSON 等。我们可以通过简单封装 noir.response 命名空间下的 content-type 函数实现。

```
(GET "/records" []
  (noir.response/content-type "text/plain" "some plain text"))
```

noir.response 命名空间下有用于处理 JSON 和 XML 的辅助函数。比如 JSON 响应，就是将内建数据结构自动转换为 JSON 字符串。

```
(GET "/get-message" []
  (noir.response/json {:message "everything went better than expected!"})
```

这个回应辅助函数非常实用，用于应对客户端发起的 Ajax 请求。

Noir API 一览

我们已经说过了，Lib-noir 提供非常多的实用特性。

cookies 命名空间提供的函数用于读写 cookie；io 命名空间提供的函数可用于访问静态资源，并且也能处理文件上传；cache 命名空间提供内容缓存的基础件；middleware 命名空间提供数个辅助函数去创建通用类型的程序 handler 和封装；最后，route 命名空间提供一个函数去创建受限路由。这有助于限制页面访问，我们放在“第 5 章 相册”来讨论这些内容。

2.5 你学到什么

在这一章，我们见识了如何通过 Clojure 搭建 Web 栈，以及一些常用程序库。我们谈及了如何与 Ring、Compojure、lib-noir 交互，通过完成比如输入验证和会话管理的任务来说明它们之间如何相互作用。

但愿你已能顺畅阅读，并理解在留言簿项目（我们在“第 1 章 起步”创建的那个项目）的代码。如果你还有疑惑，我强烈建议你去重新阅读“第 1 章”，并在 REPL 环境中尝试自己搭建这个例子。如果你还没来得及做，再提一点，借此机会把本章的例子带入留言簿程序做一遍。

在下一章，我们会使用 Liberator 建立 REST 服务。

第3章

服务组件 Liberator

在上一章中，我们谈到如何编写一个典型的 Web 应用程序，以及各组件间如何交互。例如，我们现在知道如何管理路由、编写 HTML 模板以及使用状态去管理会话。在本章中，我们来看看用别的途径来实现这种应用。

你可能已经注意到，在那个应用中客户端和服务器端并没有被割裂处理。如果我们一个不小心，会不知不觉就将客户端和服务端弄成高耦合状态。假如我们接下来准备新增不同类型的客户端，好比我们要再做一个移动端应用，那么这势必构成困扰。

本章我们将会学到如何使用 Liberator 库，以确保分离服务端和客户端的关注点。Liberator[①]是继 webmachine[②]之后又一个基于 RESTful 服务模型的 Clojure 库，这是一个受欢迎的 Erlang 服务框架。其核心特征是强调在你从前端至后端的应用中去耦合。

从概念上讲，Liberator 提供了一系列规整的方式来处理服务操作，它将收到的请求匹配资源定义中的条件和处理。使用的状态码含义描述由 HTTP RFC 2616 定义，如 200-OK、201-created、404-not found 之类。

这种方式可以便捷地实现符合标准的服务，并且有条理地组织操作。这也意味着，你的服务将自动关联和响应处理对应的 HTTP 状态码。

由于其专注将前后端逻辑分离，在处理多形式接入的互联网应用方面，Liberator

① http://clojure-liberator.github.io/liberator/
② https://github.com/basho/webmachine

确实是不错的选择。这些客户端形式不单指通用服务、单页应用，还可以是移动端应用这种非网页客户端。

3.1　创建项目

在本节我们将展开讲述如何创建一个提供静态资源的简单应用，还包括基本的会话管理和 JavaScript 对象表示法（JSON，JavaScript Object Notation）处理。

首先，我们用 compojure-app 模板来创建一个叫作 liberator-service 的应用。

```
lein new compojure-app liberator-service
```

程序工程创建完毕后，向其 project.clj 的依赖项列表中添加 Liberator 和 Cheshire[①] 这两个依赖项：

```
:dependencies
[ ...
  [liberator "0.10.0"]
  [cheshire "5.2.0"]]
```

Cheshire 是一个可以快速上手的 JSON 解析库。我们用它来解析、构造与客户端之间的请求、响应。

接下来，我们就能开始在读入-求值-打印循环（REPL，Read-evaluate-print Loop）运行位于 liberator-service.repl 命名空间的启动服务了。

当前应用程序显示的是由模板创建的默认页路由。接下来我们一起看看怎么才能用 Liberator 路由开工干活。

3.2　定义资源

Liberator 把与客户端交互行为描述成资源。这些资源是 Ring 兼容的处理，可以用在 Compojure 路由里。这些资源使用 resource 和 defresource 宏来定义。接下来打开 liberator-service.routes.home 命名空间。我们在声明中把引用的 layout 删掉，并将 resource 和 defresource 添加进来。

① https://github.com/dakrone/cheshire

```
liberator-snippets/home.clj
(ns liberator-service.routes.home
  (:require [compojure.core :refer :all]
            [liberator.core
             :refer [defresource resource request-method-in]]))
```

现在，我们可以像下面这样替换“/”路由：

```
liberator-snippets/home.clj
(defroutes home-routes
  (ANY "/" request
       (resource
        :handle-ok "Hello World!"
        :etag "fixed-etag"
        :available-media-types ["text/plain"])))
```

现在我们回头刷新一下页面，会看见页面显示：“Hello World!”。要说一下，我们已经用上 ANY，这是在资源中定义的 Compojure 路由。这将允许 Liberator 资源来处理请求类型。

如果我们要在资源中为处理命名，可以用 defresource 来代替：

```
liberator-snippets/home.clj
(defresource home
  :handle-ok "Hello World!"
  :etag "fixed-etag"
  :available-media-types ["text/plain"])

(defreoutes home-routes
  (ANY "/" request home))
```

以上这些请求路由都是简单的 map（“请求 map 的内容”详见 26 页）。

在 Liberator 应用程序接口中定义的键代表了每一个资源，具体行为进而与对应的键相关联。键可以分为如下四类：

- 判定。
- 处理。
- 行动。
- 声明。

每个键对应关联着一个常量或者函数。该函数应该接受用于描述当前上下文的唯一参数，并返回各种对应的响应。

上下文参数是个 map，其键名可以是请求、资源或者任何可选项。这些资源键名与 Ring 请求对应，并代表了当前状态，以及对上下文协商的结果。

来，我们再细看一下每种类型和其用途。

判定

判定的意图是为了找到如何处理客户端请求，这种判定键名用问号（?）结尾，并且，这种处理必须返回布尔类型。

单个判定函数返回的布尔值意味着判断的结果，要么它能返回一个 map 或 vector。假如返回的是一个 map，该判定会将结果视为真，并将该 map 合并入上下文，最终用于生成供返回的 map。若返回的是个 vector，那该集合必须包含一个布尔值，之后的内容会以 map 形式合并入返回集。

当判定出现否定结果，就会回给客户端一个对应的 HTTP 状态码。用例子来说明：如果我们需要将之前定义的路径标记为失效，就需要添加一个新的判定，其键名为 service-available?，其返回为假。

liberator-snippets/home.clj

```
(defresource home
  :service-available? false
  :handle-ok "Hello World!"
  :etag "fixed-etag"
  :available-media-types ["text/plain"])
```

这时再刷新页面，我们会看到显示收到 503 返回类型，这个状态码是告诉客户端服务不可用。

另外，我们可以使用 method-allowed?来限制资源访问，其键值为一个判定函数。

```
(defresource home
  :method-allowed?
  (fn [context]
    (= :get (get-in context [:request :request-method])))
  :handle-ok "Hello World!"
  :etag "fixed-etag"
  :available-media-types ["text/plain"])
```

由于请求检查是一种常见操作，Liberator 提供了一个键名:allowed-methods。此键

值是一个 vector，里边存放着 HTTP 方法的关键字。

```
(defresource home
  :allowed-methods [:get]
  :handle-ok "Hello World!"
  :etag "fixed-etag"
  :available-media-types ["text/plain"])
```

我们也可以将多个判定处理合并，让其使用共同资源，表示如下：

liberator-snippets/home.clj

```
(defresource home
  :service-available? true

  :method-allowed? (request-method-in :get)

  :handle-method-not-allowed
  (fn [context]
    (str (get-in context [:request :request-method]) " is not allowed"))

  :handle-ok "Hello World!"
  :etag "fixed-etag"
  :available-media-types ["text/plain"])
```

处理

处理函数需返回一个标准的 Ring 响应，处理键名都是以“handle-”前缀。当我们使用了:handle-ok 后，处理函数将返回资源中的响应。

此外还有其他的处理，例如:handle-method-not-allowed 和:handle-not-found，类似处理键的完整列表可以在官方文档页面[①]中找到。这些处理可以结合判定来返回特定响应，用于特殊的判定输出。

举例说明，如果我们想在服务不可用时，返回一个特殊响应，可做如下处理。

liberator-snippets/home.clj

```
(defresource home
  :service-available? false
  :handle-service-not-available
  "service is currently unavailable..."

  :method-allowed? (request-method-in :get)
  :handle-method-not-allowed
```

① http://clojure-liberator.github.io/liberator/doc/handlers.html

```
(fn [context]
 (str (get-in context [:request :request-method]) " is not allowed"))

:handle-ok "Hello World!"
:etag "fixed-etag"
:available-media-types ["text/plain"])
```

这样，我们的资源在判定输出时，就有自定义处理了。

操作

操作的产生来自客户端的当前状态是更新行为，比如 PUT、POST、DELETE 请求，这时的行动键名以“!”为后缀。这意味着会改变应用程序的内部状态。

一旦执行一个操作动作，我们可以使用 handle-created 处理，并将结果返回给客户端。

声明

声明是用来描述资源的能力集，例如，我们为资源添加:available-media-types，它会返回 text/plain 类型的响应。还有一种声明方式是:etag，允许客户端缓存资源。

3.3　汇总

我们先看个例子，现有一服务已有数个资源，并且允许客户端访问和储存一些数据。

应用会展示一个用户列表，并允许客户端添加用户到此列表。该客户端用 JavaScript 实现并使用 Ajax 与服务进行通信。

首先，我们先在公共目录下创建一个 HTML 静态页，并命名为 home.html。页面内容如下：

liberator-snippets/home.html

```
<html>
  <head>
    <title>Liberator Example</title>
    <script type="text/javascript"
            src="//ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.min.js">
    </script>

    <script type="text/javascript">
```

```
function renderUsers(users) {
  $('#user-list').empty();
  for(user in users)
    $('#user-list').append($('<li/>', {html: users[user]}));
  }
  function getUsers() {
    $.get("/users", renderUsers);
  }
      function addUser() {
        $.post("/add-user", {user: $('#name').val()}, renderUsers);
      }
       $(function() {getUsers();});
  </script>
</head>

<body>
  <h1>Current Users</h1>

  <ul id="user-list"></ul>
  <input type="text" id="name" placeholder="user name"/>
  <button onclick="addUser()">Add User</button>
</body>

</html>
```

该页面有几个函数，用于将给定的 JSON 数组显示为用户清单，并从/users 获取当前用户，通过/add-user 添加新用户。另外，我们还有一个显示用户列表的占位符，并且还有一个文本框和一个添加用户按钮，可以随时点击按钮来添加用户。该页面如图 3-1 所示。

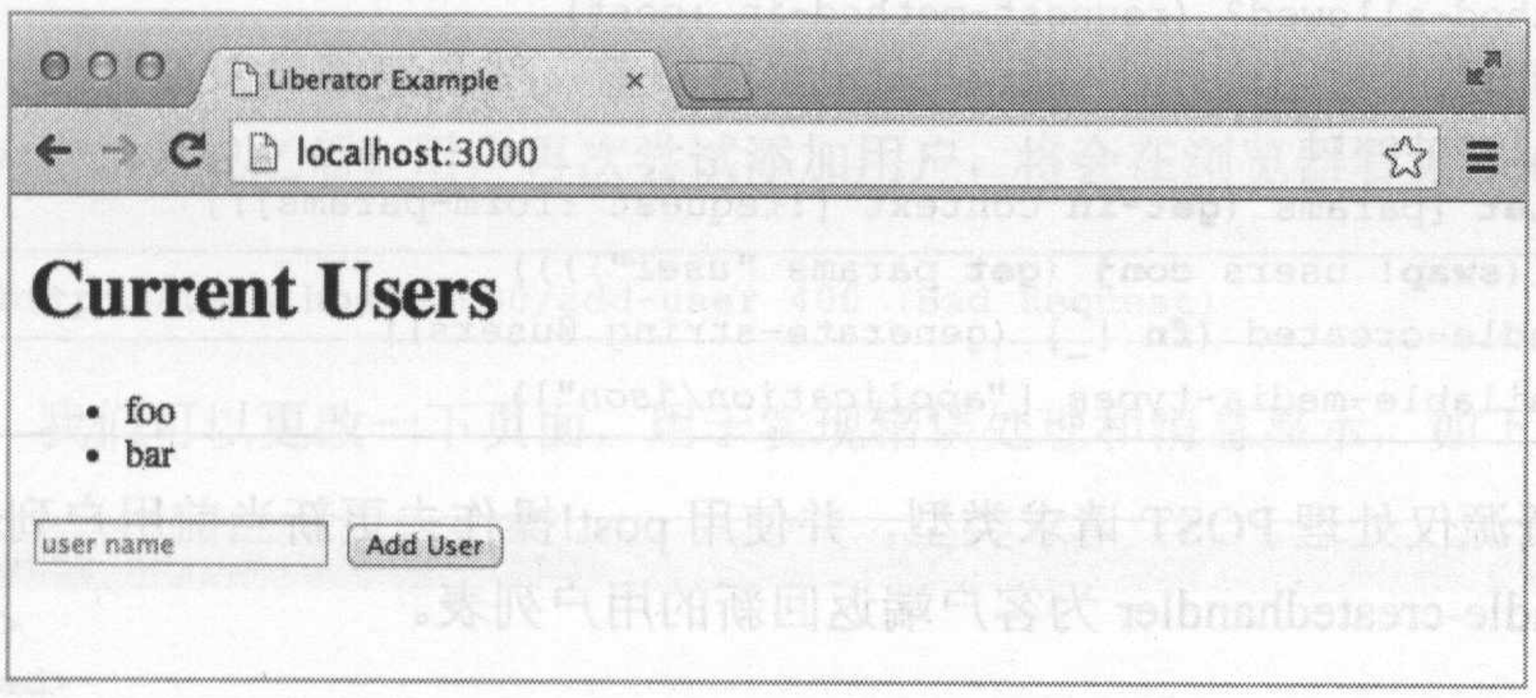

图 3-1 添加用户

现在，我们接着创建相应的资源来处理每个操作。为了能提供 JSON 数据支持，我们先在主页命名空间中添加名为 cheshire.core/generate-string 的引用：

```
(ns liberator-service.routes.home
  (:require ...
            [cheshire.core :refer [generate-string]]))
```

下一步，我们创建一个原子数据去保存用户表：

```
(def users (atom ["John" "Jane"]))
```

第一个资源将响应 GET 请求，返回一个原子类型的 JSON 数据，包含当前的所有用户。

liberator-service/src/liberator_service/routes/home.clj

```
(defresource get-users
  :allowed-methods [:get]
  :handle-ok (fn [_] (generate-string @users))
  :available-media-types ["application/json"])
```

此处的资源定义，我们使用:allowed-methods 来约束服务类型仅有 GET 请求。还使用:available-media-types 声明来说明类型为 application/json 的响应。资源最终在调用时生成 JSON 字符串来表述当前用户清单。

第二个资源用于响应 POST 请求，并将 form-params 包含的用户添加到用户列表。最终返回新的清单：

liberator-snippets/home.clj

```
(defresource add-user
  :method-allowed? (request-method-in :post)
  :post!
  (fn [context]
    (let [params (get-in context [:request :form-params])]
      (swap! users conj (get params "user"))))
  :handle-created (fn [_] (generate-string @users))
  :available-media-types ["application/json"])
```

这个资源仅处理 POST 请求类型，并使用 post!操作去更新当前用户列表。我们现在使用 handle-createdhandler 为客户端返回新的用户列表。

注意，定义的资源只是描述了细节，:handle-created 的值必须是个函数。

后面这个资源在编译时虽不会出错，但当它运行起来，你看到的仍旧是旧的用户表。这是因为决策图运行之前，(generate-string @users)已经进行求值了。

liberator-snippets/home.clj

```
(defresource add-user
  :method-allowed? (request-method-in :post)
  :post!
  (fn [context]
    (let [params (get-in context [:request :form-params])]
      (swap! users conj (get params "user"))))
  :handle-created (generate-string @users)
  :available-media-types ["application/json"])
```

如此定要确保:handle-created 键值是个函数，能在决策图运行的时候执行，我们的原生例子就是如此处理的。

你应该注意到，我们并没有阻止用户添加空账户。那么接下来，我们再为添加用户的服务环节增加请求验证。

liberator-service/src/liberator_service/routes/home.clj

```
(defresource add-user
  :allowed-methods [:post]
  :malformed? (fn [context]
                (let [params (get-in context [:request :form-params])]
                  (empty? (get params "user"))))
  :handle-malformed "user name cannot be empty!"
  :post!
  (fn [context]
    (let [params (get-in context [:request :form-params])]
      (swap! users conj (get params "user"))))
  :handle-created (fn [_] (generate-string @users))
  :available-media-types ["application/json"])
```

现在，如果账户参数值是空，就路由至 handle-mal-formed，它能告知客户端，账户名不能为空。完成之后，用户再次尝试添加用户，将会在浏览器看见 400 错误提示。

```
POST http://localhost:3000/add-user 400 (Bad Request)
```

现在，我们可以更改一下页面，用于实现错误处理和消息显示，如下所示。

liberator-snippets/home1.html

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=US-ASCII">
    <title>Liberator Example</title>

    <script type="text/javascript"
            src="//ajax.googleapis.com/ajax/libs/jquery/1.9.0/jquery.min.js">
```

```
    </script>

    <script type="text/javascript">
        function renderUsers(users) {
          $('#user-list').empty();
          for(user in users)
            $('#user-list').append($('<li/>', {html: users[user]}));
          }

          function getUsers() {
            $.get("/users", renderUsers);
          }

          function handleError(xhr) {
            $('#error').text(xhr.statusText + ": " + xhr.responseText);
          }

        function addUser() {
          var jqxhr = $.post("/add-user", {user: $('#name').val()}, renderUsers)
                       .fail(handleError);
        }

        $(function() {getUsers();});
    </script>
  </head>
  <body>
    <h1>Current Users</h1>
    <p id="error"></p>
    <ul id="user-list"></ul>
    <input type="text" id="name" placeholder="user name"/>
    <button onclick="addUser()">Add User</button>
  </body>
</html>
```

现在，如果账户文本框置空，点击添加用户按钮时，就会看到图 3-2 所示的错误提示。

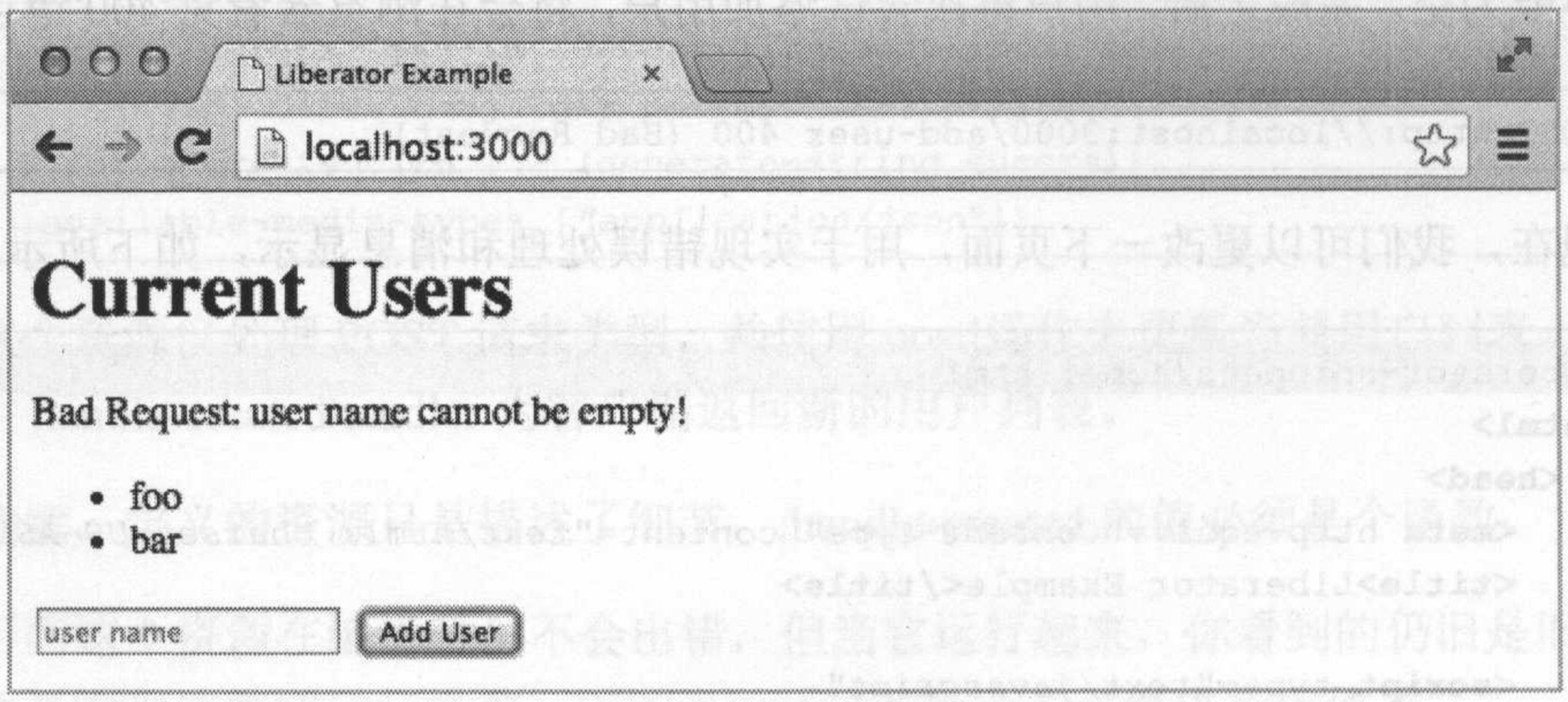

图 3-2　错误提示

在最后阶段，我们为 home.html 文件添加一个 home 资源。这样需要在 project.clj 添加 lib-noir 依赖项：

```
:dependencies [... [lib-noir "0.7.2"]]
```

除此之外，我们还要为 home 命名空间添加 clojure.java.io 和 noir.io 的引用项：

```
(ns liberator-service.routes.home
  (:require [...
            [noir.io :as io]
            [clojure.java.io :refer [file]]))
```

此刻，再创建一个名为 home 的新资源，用来维护 home.html 文件：

liberator-service/src/liberator_service/routes/home.clj

```
(defresource home
  :available-media-types ["text/html"]

  :exists?
  (fn [context]
    [(io/get-resource "/home.html")
     {::file (file (str (io/resource-path) "/home.html"))}])

  :handle-ok
  (fn [{{{resource :resource} :route-params} :request}]
    (clojure.java.io/input-stream (io/get-resource "/home.html")))
  :last-modified
  (fn [{{{resource :resource} :route-params} :request}]
    (.lastModified (file (str (io/resource-path) "/home.html")))))
```

资源会自己检查文件存在与否，并且检查文件有无更新。如果当前文件不可用，io/get-resource 会返回 nil，客户端将得到一个 404 错误提示；如果此文件从上次请求之后并无更新，客户端将会收到 304 状态码而不是文件，此举旨在告知客户端文件尚无修改。

幸亏有这种检查，只有当文件存在，并且在上次请求之后存在修改，才会真正提供服务。现在，我们添加一条路由来维护 home.html，处理方式类似默认资源：

```
(ANY "/" request home)
```

在我们的 home 命名空间中，每一段服务代码对应着一个页面，看起来大致如下：

liberator-service/src/liberator_service/routes/home.clj

```
(ns liberator-service.routes.home
  (:require [compojure.core :refer :all]
            [liberator.core :refer [defresource resource]]
            [cheshire.core :refer [generate-string]]
            [noir.io :as io]
            [clojure.java.io :refer [file]]))
(defresource home
  :available-media-types ["text/html"]

  :exists?
  (fn [context]
    [(io/get-resource "/home.html")
     {::file (file (str (io/resource-path) "/home.html"))}])

  :handle-ok
  (fn [{{{resource :resource} :route-params} :request}]
    (clojure.java.io/input-stream (io/get-resource "/home.html")))
   :last-modified
  (fn [{{{resource :resource} :route-params} :request}]
    (.lastModified (file (str (io/resource-path) "/home.html")))))

(def users (atom ["foo" "bar"]))
(defresource get-users
  :allowed-methods [:get]
  :handle-ok (fn [_] (generate-string @users))
  :available-media-types ["application/json"])

(defresource add-user
  :allowed-methods [:post]
  :malformed? (fn [context]
                (let [params (get-in context [:request :form-params])]
                  (empty? (get params "user"))))
  :handle-malformed "user name cannot be empty!"
  :post!
  (fn [context]
    (let [params (get-in context [:request :form-params])]
      (swap! users conj (get params "user"))))
  :handle-created (fn [_] (generate-string @users))
  :available-media-types ["application/json"])

(defroutes home-routes
  (ANY "/" request home)
  (ANY "/add-user" request add-user)
  (ANY "/users" request get-users))
```

如你所见，Liberator 在设计层面确保关注点分离。通过 Liberator 提供的模式思想，你只需提供一系列小巧、完备的函数，每个函数仅用于处理独立的任务。

3.4 你学到什么

至此，我们关注的是应用中服务端/客户端的交互环节。在下一章中，我们会深入学习如何连接数据库，以及通过数据库来完成工作。

第4章

访问数据库

在上一章中，我们重点关注的是处理客户端与服务端的交互，对如何实现数据持久化，仅一笔带过。在本章，我们会全面讲述怎样使用 clojure.java.jdbc 库去操纵关系型数据库来工作。届时，我们将讨论如何去编写一个简单应用，实现通过数据库记录生成 PDF 报告。

4.1 使用关系型数据库

正因为 Clojure 运行在 Java 虚拟机上，这有着得天独厚的优势，可以通过 Java 数据库连接[①]（JDBC，Java Database Connectivity）访问所有类型的数据库。我们可以轻松访问许多关系型数据库管理系统（RDBMS，Relational Database Management System），比如 MySQL、SQL Server、PostgreSQL 和 Oracle。有好些软件库都可以有效支持这些数据库，我们快看看都有哪些。

在处理关系型数据库方面，clojure.data.jdbc 是最简单的。使用这个库，你只用写下数据库的原生 SQL 语句就行了。假如你已明确了将要使用的数据库，比如 MySQL 或 PostgreSQL，这对你很可能不是什么困难。麻烦的是，如果你有数据迁移的打算，便需要重新编写查询语句来适应新的数据库。

还可以使用高抽象级的库去操纵 RDBMS，比如 SQL Korma（http://sqlkorma.com/）。这个库允许你使用领域特定语言（DSL，Domain-specific Language）写下查询语句，

① http://en.wikipedia.org/wiki/Java_Database_Connectivity

然后自己生成针对后端的数据库 SQL 语句。很明显，这种方式不再需要手动编写 SQL 语句。但有利也有弊，你不仅需要学习 DSL，而且这种方式只能访问它支持的数据库类型。在书中后面的内容，我们将看到一个使用案例。

从现在开始，我们开始关注如何使用 clojure.data.jdbc 库，它提供的都是函数式处理方式，不会为我们增加任何负担。在本书中，我们将使用 PostgreSQL 作为数据库引擎。

如果你选择使用不一样的数据库引擎，那你要意识到，那可能与你使用的 SQL 语句存在少许区别。

访问数据库

在访问数据库之前，我们首先要在 project.clj 文件中添加需要的依赖项。这里，java.jdbc 库的作用类似访问数据库的驱动。假设用的是 PostgreSQL，我们的需求中添加如下依赖项：

```
[org.clojure/java.jdbc "0.2.3"]
[postgresql/postgresql "9.1-901.jdbc4"]
```

在此处，我们为应用创建一个新的命名空间，作为数据模型部分。按惯例，将此命名空间命名为 models.db。现在，我们按之前案例（见“第 1 章 起步”）的处理方式添加 clojure.data.jdbc 引用。

```
(:require [clojure.java.jdbc :as sql])
```

下一步，我们需要去定义数据库连接。实现方式有多种途径，我们一起看看个中利弊。

定义参数 map

对定义连接而言，最简单的方法就是提供一个描述连接参数的 vector。

```
(def db {:subprotocol "postgresql"
         :subname "//localhost/my_website"
         :user "admin"
         :password "admin"})
```

这是一种常见的方式，其最大的缺点就是连接的相关信息被直接写在代码里面。假如你使用驱动直连的方式，你提供的这些参数一样受限。

指定驱动直连

还有个选择，提供一个 JDBC 数据源并进行手工配置。这种方式非常实用，如果你想指定特定驱动参数，通过惯用的参数 map 配置不太容易实现。

```
(def db
  {:datasource
    (doto (PGPoolingDataSource.)
     (.setServerName   "localhost")
     (.setDatabaseName "my_website")
     (.setUser         "admin")
     (.setPassword     "admin")
     (.setMaxConnections 10))})
```

定义 JNDI 字符串

最后，我们可以通过 Java 命名和目录服务（JNDI，Java Naming and Directory Interface）定义特殊连接，为应用服务提供的连接管理命名。

```
(def db {:name "jdbc/myDatasource"})
```

这里我们要提供一个字符串作为 JNDI 名称。在使用的应用服务上，将会配置真实的连接，并且在程序中定义一个名称，此名称必须与服务器一致。程序运行起来后，便可以开始使用查询服务了，并通过给定的名称来提供连接细节。

这一选择将环境配置与程序的代码分离开了，这当然再好不过了。例如，在服务的各个生命周期（开发、过渡、生产）可能会有所区别。你可以为不同的阶段指定不同的 JNDI 连接，并且当你面临部署应用时，配置连接细节就成了顺理成章的事了，程序代码不再需要调整，并且在构建它的时候，甚至你都不需要记着去保存配置文件或者环境配置。

现在，我们已经有一个数据库连接了，让我们看看怎么通过它完成一些基本任务吧。每个数据库操作都必须使用 with-connection 宏进行封装。这个宏确保在函数执行完毕时清理连接。

创建表

我们使用 create-table 函数去创建表，需要提供表名称、列名称以及它们的类型。我们来写一个函数去创建一个表，用来保存用户记录，每条记录拥有一个 ID 和密码。

```
(defn create-users-table []
  (sql/with-connection db
    (sql/create-table
     :users
      [:id "varchar(32) PRIMARY KEY"]
      [:pass "varchar(100)"])))
```

这里，调用 create-table 用于创建一个新的用户表。宏的第一个参数是具体的表名称，其后的参数是用来表达列的 vector。每一个列的格式是[:name type]，这里:name 是列名，type 可以是 SQL 字符串或类似:int、:boolean、:timestamp 的关键字。注意，列名称不能包含“-”，因为这在 SQL 语法中不合法。

查询记录

从数据库查询记录，我们使用 with-query-results 宏。它接受一个包含 SQL 表达式的 vector，随后是它的参数，返回的结果是个惰性序列。这使得我们不必将整个结果加载到内存，就能使用返回的数据。

但也由于结果是惰性序列，如果我们需要函数返回此结果，则必须在返回之前对结果进行求值。假如不需要，会在离开 with-connection 函数时关闭连接，到时候再对结果求值，只能得到 nil。我们可以使用 doall 强迫对其求值。如果我们按如下代码简单查询一个元素，其实最后也暗含了对结果进行求值。

```
(defn get-user [id]
  (sql/with-connection db
    (sql/with-query-results
      res ["select * from users where id = ?" id] (first res))))
```

在这段代码中，我们创建一个接受用户 ID 的函数，并得第一项返回。

注意，通过在参数后面指定一个包含声明语句的 vector，用来实现参数化查询。这种方式通常用于防范 SQL 注入攻击。

插入记录

对于将数据插入数据库，可选的方式有很多。如果你有一个 map，其键名对应表的列名，那么你可以使用 insert-record 函数简单处理。

```
(defn add-user [user]
  (sql/with-connection db
    (sql/insert-record :users user)))

(add-user {:id "foo" :pass "bar"})
```

如果你想同时插入多条记录，你可以使用 insert-records 取代之。

```
(sql/with-connection db
  (sql/insert-records
    :users
    {:id "foo" :pass "x"}
    {:id "bar" :pass "y"}))
```

我们同样也可以使用 insert-rows 函数去指定记录的值。

```
(defn add-user [id pass]
  (sql/with-connection db
    (sql/insert-rows :users
      [id pass])))
```

此函数指定一个 vector 表示在数据表中定义的列项的值。如果我们仅插入部分行，我们可以使用 insert-values 取代之。

```
(sql/insert-values :users [:id] ["foo"])
```

第一个参数是表名称，随后是一个 vector 指定要更新的列名。最后一个 vector 包含列的值。

更新现有记录

对于修改已有记录，可以使用 update-values 和 update-or-insert-values 函数。其第一步是确认数据库记录是否存在，有则试图进行更新，没有就插入一条新记录。

```
(sql/update-values
  :users
  ["id=?" "foo"]
  {:pass "bar"})

(sql/update-or-insert-values
  :users
  ["id=?" "foo"]
  {:pass "bar"})
```

删除记录

从数据库删除记录，我们可以使用 delete-rows 函数：

```
(sql/delete-rows :users ["id=?" "foo"])
```

事务

当我们需要同时运行多条语句，并且还要确保所有语句都会成功执行时，我们使用事务来处理。

一旦有语句抛出异常，那么事务便会将刚才修改的内容回滚至事务执行之前。

```
(sql/with-connection db
  (sql/transaction
    (sql/update-values
      :users
      ["id=?" "foo"]
      {:pass "bar"})

    (sql/update-values
      :users
      ["id=?" "bar"]
      {:pass "baz"})))
```

4.2 生成报表

如何轻松生成报表？本节我们学习使用 clj-pdf 库[①]，它能很方便地实现用数据库获取的数据生成报表。

① https://github.com/yogthos/clj-pdf

假设我们的应用程序会有个雇员表，其中填充了一些简单数据。我们使用这些数据创建几份不一样的 PDF 报告，并且允许用户选择希望浏览的报告类型。

我们首先得配置数据库，先举个例子，假如我们使用的数据库是 PostgreSQL。

配置 PostgreSQL 数据库

安装 PostgreSQL 非常简单。如果你是用 OS X，那么你可以直接运行 Postgres.app[①]。在 Linux 上，你可以通过包管理安装 PostgreSQL。比如在 Ubuntu 上，直接运行“sudo apt-get install postgresql”。

一旦安装成功，在控制台输入 psql 命令打开 psql 命令行，通过命令行设置 postgres 密码。

```
sudo -u postgres psql postgres
\password postgres
```

完成了默认账户的设置之后，我们再创建一个 admin 账户并为其设置密码。

```
CREATE USER admin WITH PASSWORD 'admin';
```

接下来，通过执行以下命令，我们可以创建一个叫作 REPORTING 的数据库实例，用于储存我们的报告：

```
CREATE DATABASE REPORTING OWNER admin;
```

注意，我们在这里使用 admin 用户是为了节省时间。你经常需要为产品创建并授权必要权限的专用账户去管理数据库。

我们用这种方式配置了数据库，接下来，我们使用 compojure-app 模板创建一个名为 reporting-example 的新应用。

现在打开 project.clj 文件，为其添加必要的依赖项：

```
:dependencies [...
               [postgresql/postgresql "9.1-901.jdbc4"]
               [org.clojure/java.jdbc "0.2.3"]
               [clj-pdf "1.11.6"]
```

① http://postgresapp.com/

启动 REPL，运行 reporting-example.repl 命名空间下的(start-server)。

当 REPL 运行起来，我们新创建一个名为 reporting-example.models.db 的命名空间，在此添加数据库配置。

我们打开 db 命名空间，使用 clojure.java.jdbc 创建数据库连接。

```
reporting-example/src/reporting_example/models/db.clj
(ns reporting-example.models.db
  (:require [clojure.java.jdbc :as sql]))

(def db {:subprotocol "postgresql"
         :subname "//localhost/reporting"
         :user "admin"
         :password "admin"})
```

接下来，我们创建员工表并且使用一些简单数据填充：

```
reporting-example/src/reporting_example/models/db.clj
(defn create-employee-table []
  (sql/create-table
    :employee
    [:name "varchar(50)"]
    [:occupation "varchar(50)"]
    [:place "varchar(50)"]
    [:country "varchar(50)"]))

(sql/with-connection
  db
  (create-employee-table)
  (sql/insert-rows
    :employee
    ["Albert Einstein", "Engineer", "Ulm", "Germany"]
    ["Alfred Hitchcock", "Movie Director", "London", "UK"]
    ["Wernher Von Braun", "Rocket Scientist", "Wyrzysk", "Poland"]
    ["Sigmund Freud", "Neurologist", "Pribor", "Czech Republic"]
    ["Mahatma Gandhi", "Lawyer", "Gujarat", "India"]
    ["Sachin Tendulkar", "Cricket Player", "Mumbai", "India"]
    ["Michael Schumacher", "F1 Racer", "Cologne", "Germany"]))
```

最后，我们编写一个函数用于从表中读取记录。

```
reporting-example/src/reporting_example/models/db.clj
(defn read-employees []
  (sql/with-connection db
    (sql/with-query-results rs ["select * from employee"] (doall rs))))
```

现在运行 read-employees 确保一切按预期运行。我们会看到如下内容。

```
(read-employees)
({:country "Germany",
  :place "Ulm",
  :occupation "Engineer",
  :name "Albert Einstein"}
  {:country "UK",
  :place "London",
  :occupation "Movie Director",
  :name "Alfred Hitchcock"}
  ...)
```

你应该注意到，read-employees 的结果是个包含 map 的列表，map 的键名是数据库表的一系列名称。

我们来看看如何在数据库使用它创建一个员工表单。

生成报表

clj-pdf 库的语法类似于 Hiccup，将元素定义在文档中。文档本身由 vector 表示。文档 vector 必须包含一个 map，map 的第一个元素为本身的元信息，其后是一个或多个元素用以表示文档的内容。

我们现在创建一个名为 reporting-example.reports 的命名空间，并通过几个例子来创建 PDF 文档。我们使用 pdf 函数去创建报告，并且用 template 函数去格式化输入的数据。

```
(ns reporting-example.reports
  (:require [clj-pdf.core :refer [pdf template]]))
```

pdf 函数接受两个参数。第一个参数，可以是表示文档的 vector，也可以是一个能读取元素的输入流。第二个参数用于表示输出文件名字符串或是一个输出流。

现在生成我们的第一个 pdf，在我们的 reports 命名空间运行以下内容：

```
(pdf
  [{:header "Wow that was easy"}
   [:list
    [:chunk {:style :bold} "a bold item"]
```

```
    "another item"
    "yet another item"]
   [:paragraph "I'm a paragraph!"]]
  "doc.pdf")
```

如你所见，报告由 vector 构成，第一个是一个表示元素类型的关键字，其后是可选的元信息和内容。在前面的报告中，我们有一个包含三行内容的列表，其后是另一个段落。在我们项目的根目录，PDF 会写入一个名为 doc.pdf 的文件。文件内容看上去如图 4-1 所示。

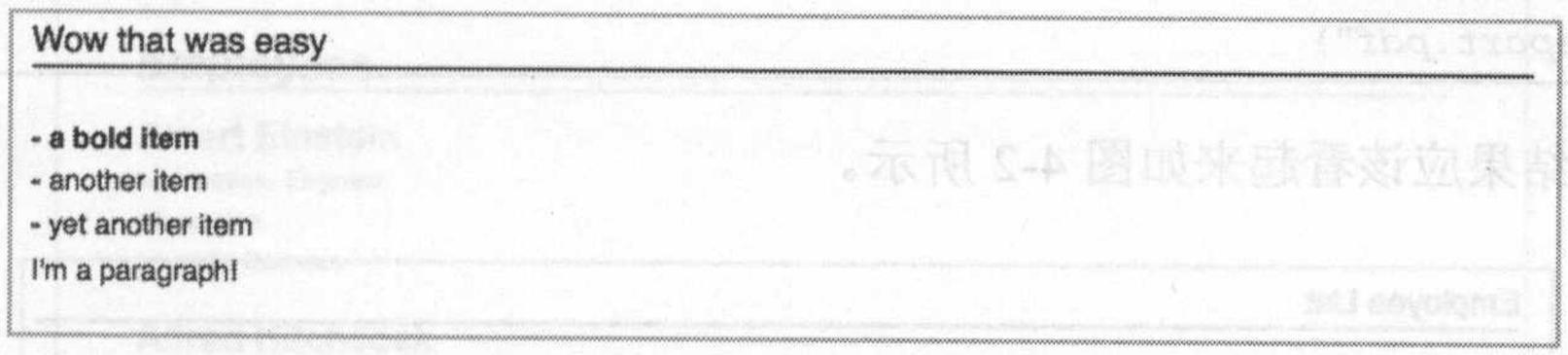

图 4-1 我们的第一个 PDF

接下来，我们来看看怎样使用 template 宏去格式化员工信息，让表单美观。这个宏的参数是一个集合，其中的元素是$前缀的名称，能从数据中提取同名键值。

template 返回一个函数，此函数接受一个 map 的序列，并且适用于提供的模板，对于序列的每一个元素。在我们的实例中，由于我们正在创建一张表，这个模板就是简单的一个 vector，其中的名字是每一行单元格的键名。我们接下来在 reporting-example.reports 命名空间添加如下模板：

```
(def employee-template
  (template [$name $occupation $place $country]))
```

我们在 db 命名空间添加引用，然后尝试针对数据库运行我们的模板：

```
reporting-example/src/reporting_example/reports.clj
(ns reporting-example.reports
  (:require [clj-pdf.core :refer [pdf template]]
            [reporting-example.models.db :as db]))
```

在 REPL 运行了(employee-template (take 2 (db/read- employees)))之后，我们应该看到以下输出：

```
(["Albert Einstein" "Engineer" "Ulm" "Germany"]
 ["Alfred Hitchcock", "Movie Director", "London", "UK"])
```

模板运行结果看起来比较满意。我们快来使用它去生成包含所有员工清单的完整报表吧。

```
(pdf
 [{:header "Employee List"}
  (into [:table
         {:border false
          :cell-border false
          :header [{:color [0 150 150]} "Name" "Occupation" "Place" "Country"]}]
        (employee-template (db/read-employees)))]
 "report.pdf")
```

报表结果应该看起来如图 4-2 所示。

Employee List

Name	Occupation	Place	Country
Albert Einstein	Engineer	Ulm	Germany
Alfred Hitchcock	Movie Director	London	UK
Wernher Von Braun	Rocket Scientist	Wyrzysk	Poland
Sigmund Freud	Neurologist	Pribor	Czech Republic
Mahatma Gandhi	Lawyer	Gujarat	India
Sachin Tendulkar	Cricket Player	Mumbai	India
Michael Schumacher	F1 Racer	Cologne	Germany

图 4-2　员工清单报表

没错，我们生成这个报表用的模板着实不给力。那我们现在来看看另一个例子。这里我们会将数据输出到列表，并且对每个元素风格化：

reporting-example/src/reporting_example/reports.clj

```
(def employee-template-paragraph
  (template
   [:paragraph
    [:heading {:style {:size 15}} $name]
    [:chunk {:style :bold} "occupation: "] $occupation "\n"
    [:chunk {:style :bold} "place: "] $place "\n"
    [:chunk {:style :bold} "country: "] $country
    [:spacer]]))
```

现在，我们使用 employee-template-paragraph 来创建报表：

```
  (pdf
    [{}
     [:heading {:size 10} "Employees"]
     [:line]
     [:spacer]
     (employee-template-paragraph (db/read-employees))]
    "report.pdf")
```

我们的新报表应该会如图 4-3 所示。

Employees

Albert Einstein
occupation: Engineer
place: Ulm
country: Germany

Alfred Hitchcock
occupation: Movie Director
place: London
country: UK

Wernher Von Braun
occupation: Rocket Scientist
place: Wyrzysk
country: Poland

Sigmund Freud
occupation: Neurologist
place: Pribor
country: Czech Republic

Mahatma Gandhi
occupation: Lawyer
place: Gujarat
country: India

Sachin Tendulkar

图 4-3 员工清单表

显示报表

现在，我们已经使用数据创建了几份报表，接下来再看看如何将其用于我们的应用。我们将会使用之前的示例创建函数来生成清单并且制作报表。

reporting-example/src/reporting_example/reports.clj

```
(defn table-report [out]
  (pdf
    [{:header "Employee List"}
     (into [:table
            {:border false
```

```
                :cell-border false
                :header [{:color [0 150 150]} "Name" "Occupation" "Place" "Country"]}]
              (employee-template (db/read-employees)))]
    out))

(defn list-report [out]
  (pdf
   [{}
    [:heading {:size 10} "Employees"]
    [:line]
    [:spacer]
   (employee-template-paragraph (db/read-employees))]
   out))
```

接下来，打开 reporting-example.routes.home 命名空间，为生成报表路由添加需要的依赖项。

reporting-example/src/reporting_example/routes/home.clj

```
(ns reporting-example.routes.home
  (:require [hiccup.element :refer [link-to]]
            [ring.util.response :as response]
            [compojure.core :refer [defroutes GET]]
            [reporting-example.reports :as reports]
            [reporting-example.views.layout :as layout]))
```

我们更新 home 函数，为每一个报表提供链接：

reporting-example/src/reporting_example/routes/home.clj

```
(defn home []
  (layout/common
    [:h1 "Select a report:"]

    [:ul
     [:li (link-to "/list" "List report")]
     [:li (link-to "/table" "Table report")]]))
```

现在，我们写个函数去生成响应。我们创建一个输入流，使用其提供的字节数组并设为响应。我们一样要设置头信息 content-type、content-disposition 及负载长度。

reporting-example/src/reporting_example/routes/home.clj

```
(defn write-response [report-bytes]
  (with-open [in (java.io.ByteArrayInputStream. report-bytes)]
  (-> (response/response in)

      (response/header "Content-Disposition" "filename=document.pdf")
      (response/header "Content-Length" (count report-bytes))
      (response/content-type "application/pdf")) ))
```

我们再写个函数用来生成报告。这个函数会创建一个 ByteArrayOutputStream 用于保存报表。这时它会调用 report-generation 的其中一个函数。一旦报表生成完毕，我们就可以将输出流的内容压给 write-response。

reporting-example/src/reporting_example/routes/home.clj

```
(defn generate-report [report-type]
  (try
    (let [out (new java.io.ByteArrayOutputStream)]
      (condp = (keyword report-type)
        :table (reports/table-report out)
        :list (reports/list-report out))
      (write-response (.toByteArray out)))

    (catch Exception ex
      {:status 500
       :headers {"Content-Type" "text/html"}
       :body (layout/common
               [:h2 "An error has occured while generating the report"]
               [:p (.getMessage ex)])})))
```

最后，我们还需要为报表创建一个新的路由来提供服务。

reporting-example/src/reporting_example/routes/home.clj

```
(defroutes home-routes
  (GET "/" [] (home))
  (GET "/:report-type" [report-type] (generate-report report-type)))
```

现在，你用浏览器打开 http://localhost:3000，选择其中一条报表的链接。当你点击链接，就应该能获取对应的报表。

4.3 你学到什么

本章包含了使用关系型数据库的一些基本知识。你现在应该已经学会了如何使用基本的数据库操作，并且也见识了一个报表应用。通过本章，我们能看出来，数据库记录能直接映射为 Clojure 的数据结构，所以，Clojure 社区认为对象-关系映射库不是太有必要。

下一章，我们将集目前所学，编写一个相册应用。

第5章

相册

在这一章，我们会集合前面章节的知识来创建一个相册应用。

5.1 开发流程

我们首先针对应用的工作流，创建一个大致的结构，并将其简单罗列出来，接下来再对每一个过程的细节逐步完善。

我们通过 REPL 实现交互式程序开发。这将有助于我们增量式开发应用，不需要为了运行新写下的函数而重启环境。

在 REPL 里面尝试时不用有所顾虑。打个比方，假如你为了从数据库取数据而编写一个功能函数，就先在 REPL 中试试，看看它的结果数据，运行通过了再决定录用。

5.2 相册有什么

我们要做的网站通过鉴别不同的用户，为用户提供不同的页面，并围绕这些页面展开。我们来罗列一份关于用户可能的行为清单。每一个独立的行为就是一条有别于其他行为的特殊工作流，需要独立完善其功能。

1. 账户注册

为将内容上传至网站，用户必须在网站拥有账户。为实现这一点，我们必须有一

个页面，用来收集用户的详细信息以实现注册，比如 ID、密码，并且对其进行验证，为此用户在数据库创建一个条目。

2. 登录登出

一旦用户创建了一个账户，他便应该能通过凭证登录了。假如用户的会话信息中还没有账户信息，我们需要在页面上显示一个登录表格。如果用户登录成功，我们还希望能显示一个登出按钮。

3. 上传图片

解决了前面这些事情之后，我们将注意力放在添加核心功能上。首先，我们需要为用户提供某种途径，用于将图片上传至网站。上传完毕后，我们还要为其创建缩略图，用于在相册中展示出来。

4. 显示图片

当我们完成图片上传之后，还需要显示它们。我们在列表中显示缩略图并且提供链接访问全尺寸图片。由于我们的网站是多用户的，我们就需要一种途径来展开用户相册。这样，访问者就能浏览来自所有用户的内容。

5. 删除图片

用户可能希望删除一些他们自己上传的图片，那么我们一样也要提供实现删除的接口。当用户决定要删除图片时，他首先需要以某种途径来选定它们，并且要在服务端删除图片原件和缩略图。

6. 删除账户

用户还有可能想删除自己的账户，虽然，我们希望这种情况永远不会发生，但是我们还是要考虑到这种情况的发生。当用户删除之后，我们需要从数据库中将用户移除，并且删除用户提交的所有照片和对应的缩略图。

代码架构

既然我们已经明确了要做的内容，那么接下来就要想想该怎么做了，现在我们开始思考用什么手段来实现。为了方便管理，我们为工作流创建独立的命名空间。将所有有关的操作放在同一个命名空间下以方便划分逻辑关系。

为理解我们程序的本质，我们先弄清数据模型。搞清楚我们需要收集什么数据，以及这些数据怎样帮我们弄清工作流和用户用例。因此，创建应用的第一步是配置数据库并创建必要的数据表。

如果你使用面向对象语言的 Web 框架开发，不管你是手写 SQL 语句，还是使用对象关系映射框架（如 Hibernate[①]）来实现，你很可能在惯例上首先去创建对象模型，并将其映射到数据库。

在我们的程序中，数据库就是我们的数据模型。因为 Clojure 的数据和逻辑是分开处理的，不会出现从数据库复制数据用于不同的数据结构。根据这种原则，我们要定义的数据表自然是程序的数据模型。在后续章节，我们还会涉及如何使用原生的 Clojure 领域特定语言来对数据库实现存取操作。

5.3　创建应用程序

为创建应用，我们只需打开 Eclipse，选择创建一个新的 Leiningen 工程即可。接下来设置项目名称并修改由 compojure-app 模板提供的默认配置（如有必要，可以参看“第 1 章 起步”中更多关于本阶段的详细步骤）。我们使用 PostgreSQL 作为我们的数据库，这样我们还需为项目添加必要的依赖项。一旦项目初始化完毕，就打开 project.clj，在依赖项列表中添加 postgresql、clojure.java.jdbc、lib-noir：

picture-gallery-a/project.clj

```
[postgresql/postgresql "9.1-901.jdbc4"]
[org.clojure/java.jdbc "0.2.3"]
[lib-noir "0.7.6"]
```

因为我们用到了 lib-noir，还需要为 handler 添加它的中间件，才能正常工作。我们打开 picture-gallery.handler 命名空间，为其添加少许内容。首先我们需要引用我们将会用到的库。在代码页的顶部，也就是:require 项中添加：

```
(:require ... [noir.util.middleware :as noir-middleware])
```

app-handler 中间件分布在 noir.middleware 命名空间下，我们用其为网站建立 handler。app-handler 可以为我们建立所有的常用中间件，如会话管理之类。

① http://www.hibernate.org/

我们通过如下代码替换定义的 app：

```
(def app (noir-middleware/app-handler [home-routes app-routes]))
```

我们现在可以从命名空间中移除一些引用，因为这些已由 app-handler 接管。命名空间声明现在应该看起来如下：

```
(ns picture-gallery.handler
  (:require [compojure.route :as route]
            [compojure.core :refer [defroutes]]
            [noir.util.middleware :as noir-middleware]
            [picture-gallery.routes.home :refer [home-routes]]))
```

配置 handler 的同时，我们将包管理导航至 picture-gallery.repl 命名空间并运行。这样就启动了 REPL，在这里运行 start-server 函数，程序就跑起来了。接下来，你能在 REPL 中看到如下内容输出：

```
;; Clojure 1.5.1
=> (start-server)
picture-gallery is starting
Started server on port 3000
You can view the site at http://localhost:3000
```

程序启动之后，会弹出一个浏览器窗口，指向 localhost:3000，如图 5-1 所示，显示由模板创建的主页面。

图 5-1 默认页

5.4 程序数据模型

我们现在准备开始创建前面所说的应用程序，由于定义数据模型是其后所有任务的先决条件，我们优先处理它。

配置数据库

我们在当前数据库中新建一个名为 gallery 的实例。在 PostgreSQL，创建一个管理员账户（需配置密码）用于管理，接下来运行如下命令，为我们的程序添加一个实例。

```
CREATE DATABASE GALLERY OWNER admin;
```

现在，我们的数据库已经可以开始使用了，我们来看看怎么连接。我们可以使用 clojure.java.jdbc 去建立 Java 数据库连接（JDBC，Java Database Connectivity）。

我们在 models 下面创建一个名为 db 的命名空间，然后再次建立连接数据库。为简单起见，我们使用在第 4 章（“访问数据库”，第 67 页）介绍的第一种方法定义数据库连接。这种方式要求我们按 clojure.java.jdbc 的要求，将连接参数放在一个 map 中。

picture-gallery-a/src/picture_gallery/models/db.clj

```
(ns picture-gallery.models.db
  (:require [clojure.java.jdbc :as sql]))

(def db
  {:subprotocol "postgresql"
   :subname "//localhost/gallery"
   :user "admin"
   :password "admin"})
```

定义数据模型

当连接建立后，我们可以考虑定义必要的数据表。我们新建一个名为 picture-gallery.models .schema 的命名空间。这个命名空间会保存表定义，并且也能作为我们模型的文档。

picture-gallery-a/src/picture_gallery/models/schema.clj

```
(ns picture-gallery.models.schema
  (:require [picture-gallery.models.db :refer :all]
            [clojure.java.jdbc :as sql]))
```

我们的第一个任务是实现用户注册。作为第一步，我们应该尝试用模型来储存用户信息。

我们先编写一个函数，用于新建数据表单，用来保存用户账户。这个数据表将定义用户记录，此记录将贯穿我们整个程序。

每个用户都拥有一个 ID 和密码，两者皆为字符串。由于将会对密码进行哈希处理，所以我们应该考虑让此字段足够长。

由于用户 ID 代表了独一无二的账户，我们将其设为主键正合适。这将确保不创建重复 ID。

picture-gallery-a/src/picture_gallery/models/schema.clj

```
(defn create-users-table []
  (sql/with-connection db
    (sql/create-table
      :users
      [:id "varchar(32) PRIMARY KEY"]
      [:pass "varchar(100)"])))
```

我们在 REPL 中运行 create-users-table 函数以创建数据表。成功之后，你应该会看见如下输出：

```
#<Namespace gallery.models.schema>
(0)
```

我们的用户表已经就位，接下来应该着手进行第一个任务，也就是显示注册页，并且为用户提供创建账户的入口。

5.5 任务 1：账户注册

用户注册和验证功能与程序的其他功能相比，是相对独立的工作流。将两者的工作流放在一个命名空间将非常合适，容易理解也不易混淆。我们新建一个名为 picture-gallery.routes.auth 的命名空间，并将账户验证函数和路由处理都放在这里，接下来，实现这些函数去处理这个任务。

在动手写代码之前，我们需要理解在用户注册环节所涉及的工作流。用户需要在表单中键入一些标识信息。表单提交的内容先通过 handler 处理之后，才能决定是否要创建用户账户。

其实用户信息内容并不复杂，我们可以一步到位，开辟一个独立的页面，让用户填写一张表单来提交信息。表单最后被提交到服务端，告知 handler 验证输入内容并创建账户。

不必多说，第一步，我们首先在命名空间声明处引用相关的库。

```
(ns picture-gallery.routes.auth
  (:require [hiccup.form :refer :all]
            [compojure.core :refer :all]
            [picture-gallery.routes.home :refer :all]
            [picture-gallery.views.layout :as layout]
            [noir.session :as session]
            [noir.response :as resp]))
```

接下来，我们要做的类似在第 2 章（“Clojure 的 Web 栈”，第 23 页）的处理办法。账户注册的目的是为了收集用户信息，我们希望在数据库创建一条记录对信息保存。我们现在需要收集的内容是用户 ID、密码以及确认密码（为了确保密码输入正确而重复键入的内容）。让我们为其创建表单并添加一个控制器桩吧。

为验证用户，我们首先为其定义路由，其 GET 路由用于显示页面，POST 路由用于处理表单提交内容。

picture-gallery-a/src/picture_gallery/routes/auth.clj

```
(defroutes auth-routes
  (GET "/register" []
       (registration-page))
  (POST "/register" [id pass pass1]
        (handle-registration id pass pass1)))
```

在路由就位之后，我们开始编写具体的函数来展示注册页面及处理注册过程。回忆一下，我们曾定义过用户表单，并添加过账号密码栏。我们可以为表单中的文本框用同样的名称。

```
(defn registration-page [& [id]]
  (layout/common
    (form-to [:post "/register"]
             (label "user-id" "user id")
             (text-field "id" id)
             [:br]
             (label "pass" "password")
             (password-field "pass")
             [:br]
             (label "pass1" "retype password")
             (password-field "pass1")
             [:br]
             (submit-button "create account"))))
(defn handle-registration [id pass pass1]
  (session/put! :user id)
  (resp/redirect "/"))
```

现在，我们已经定义了路由并写下了处理函数，我们还需将其暴露在我们的处理中。首先，我们需要在 picture-gallery.handler 命名空间中引用 picture-gallery.routes.auth，为加载我们的新路由，还需更新 app 的定义。

```
(:require ...
          [picture-gallery.routes.auth :refer [auth-routes]])

(def app (noir-middleware/app-handler
          [auth-routes
          home-routes
          app-routes]))
```

在浏览器中打开 localhost:3000/register，可以看到图 5-2 所示的内容。

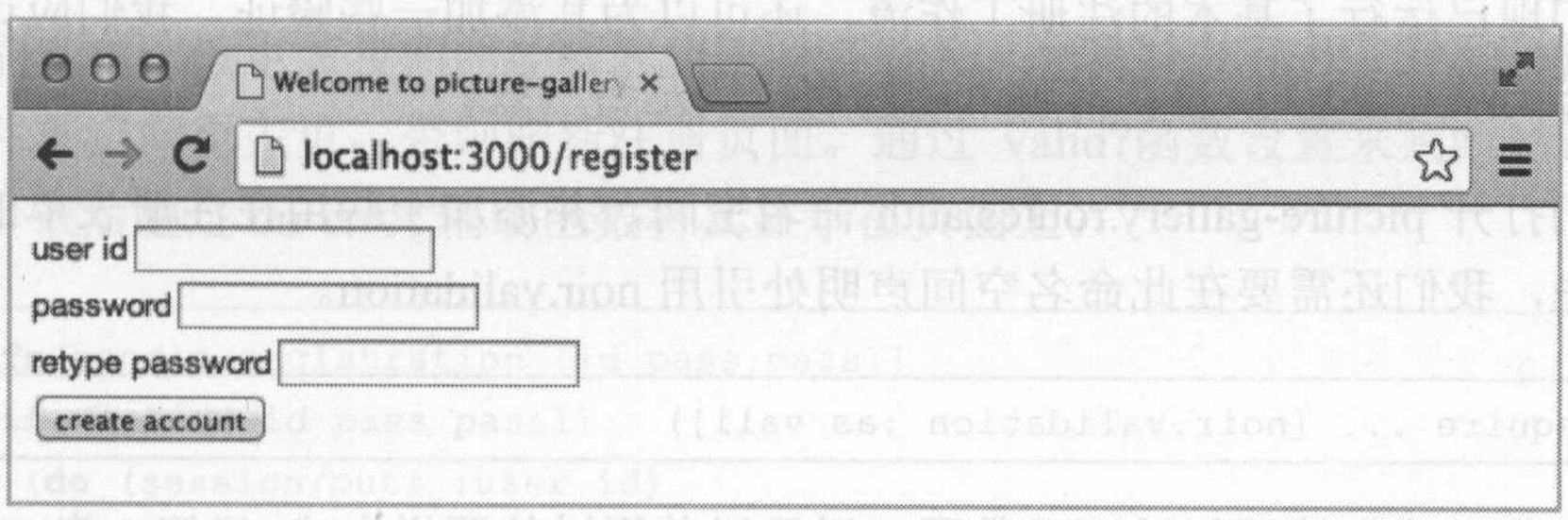

图 5-2 注册页面

需要注意的是，REPL 偶尔也会莫名其妙地出错。这通常是由错误代码导致命名空间编译失败导致的。在这种情况下，失效版本的代码也可能可以运行。如果你察觉到出现了这种现象，最简单的办法就是重启 REPL。这会让所有命名空间重新编译，假如出现错误，你将会在控制台看到出错堆栈信息。

当加载页面时，我们可以检查能否向服务端提交用户详细信息。我们现在可以修改在 picture-gallery.routes.home 命名空间的 home 函数，用我们的内容取代生成的样板页面。

首先，如果会话中存在有效的用户信息，我们就在欢迎内容中显示用户 ID。要实现这些，我们首先需要引用 noir.session 命名空间。

```
(ns picture-gallery.routes.home
  (:require ... [noir.session :as session]))
```

接下来，我们通过从会话中简单获取:user 的值，并将其显示在页面上。

```
(defn home []
  (layout/common [:h1 "Hello " (session/get :user)]))
```

现在，我们导航至 localhost:3000/register，填写用户详细信息，并点击“create account”（创建账户）按钮。我们应该被带到主页，并且会在欢迎信息中看到用户 ID，如图 5-3 所示。

图 5-3　主页

我们现已运行了基本的注册工作流，还可以为其添加一些验证。我们应该检查用户没有忘记填写 ID，以及比对输入的两次密码是否一致。

我们打开 picture-gallery.routes.auth 命名空间，并添加一些用在注册表单的验证函数。首先，我们还需要在此命名空间声明处引用 noir.validation。

```
(:require ... [noir.validation :as vali])
```

下一步，我们该添加验证函数了，以及以曾用过的同样方式（“第 2 章 Clojure 的 Web 技术栈”，第 23 页）添加名为 error-item 的格式化函数。

picture-gallery-a/src/picture_gallery/routes/auth.clj

```
(defn valid? [id pass pass1]
  (vali/rule (vali/has-value? id)
             [:id "user id is required"])
  (vali/rule (vali/min-length? pass 5)
             [:pass "password must be at least 5 characters"])
  (vali/rule (= pass pass1)
             [:pass "entered passwords do not match"])
  (not (vali/errors? :id :pass :pass1)))

(defn error-item [[error]]
  [:div.error error])
```

现在，更新注册页面用来验证输入并处理对应的响应。

```
(defn registration-page [& [id]]
  (layout/common
    (form-to [:post "/register"]
```

```
(vali/on-error :id error-item)
(label "user-id" "user id")
(text-field "id" id)
[:br]
(vali/on-error :pass error-item)
(label "pass" "password")
(password-field "pass")
[:br]
(vali/on-error :pass1 error-item)
(label "pass1" "retype password")
(password-field "pass1")
[:br]
(submit-button "create account"))))
```

我们现在可以更新我们的控制器用于验证输入，如果验证通过，我们将用户添加进会话并重定向至主页，否则刷新注册页面。通过 valid?函数设置来判断错误并置错误状态，最后通过 on-error 辅助函数将其显示在页面上。

```
(defn handle-registration [id pass pass1]
  (if (valid? id pass pass1)
    (do (session/put! :user id)
      (resp/redirect "/"))
    (registration-page id)))
```

我们来试试提交不完整的注册表单，用以测试我们的验证规则现在能不能检查(如图 5-4 所示)。

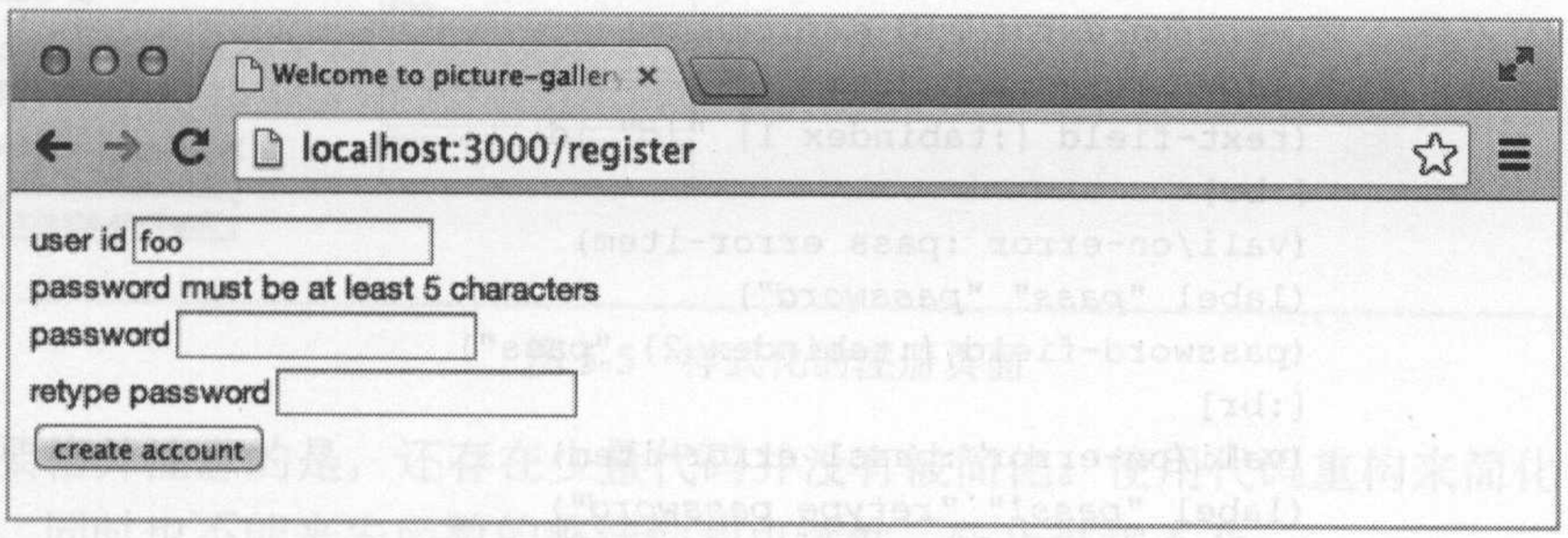

图 5-4 注册页面错误

我们还能为页面添加一些样式，使其看起来更美观。我们打开由模板生成的 screen.css 文件，此文件放在 resources/public/css 文件夹下。你应该注意到了，里面已经定义了一些 CSS。现在，我们在已有内容中添加一些自己的样式。

picture-gallery-a/resources/public/css/screen.css

```
body {
    background-color: #fff;
    color: #555;
    font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
    font-size: 13px;
}
h1 {
    text-align: center;
}
label {
    width:150px;
    float:left;
}
.error {
    color: red;
}
```

现在，表单将会适当对齐，而且，错误内容会显示为红色。我们这次不会过多关注样式，因为随着网站功能逐渐完善，我们的页面元素还会发生变化。直到我们的页面定型，我们再回头考虑页面样式。

我们要添加给表单的另一个特性是输入栏的标签索引（Tab Indexes）。这使得用户能通过 Tab 键切换输入栏。

```
(defn registration-page [& [id]]
  (layout/common
    (form-to [:post "/register"]
             (vali/on-error :id error-item)
             (label "user-id" "user id")
             (text-field {:tabindex 1} "id" id)
             [:br]
             (vali/on-error :pass error-item)
             (label "pass" "password")
             (password-field {:tabindex 2} "pass")
             [:br]
             (vali/on-error :pass1 error-item)
             (label "pass1" "retype password")
             (password-field {:tabindex 3} "pass1")
             [:br]
             (submit-button {:tabindex 4} "create account"))))
```

这个函数有一些重复内容。这明显是需要我们对其重构的。我们来写一个辅助函数来添加所有的元素。

picture-gallery-a/src/picture_gallery/routes/auth.clj

```
(defn control [id label field]
  (list
    (vali/on-error id error-item)
    label field
    [:br]))
(defn registration-page [& [id]]
  (layout/common
    (form-to [:post "/register"]
             (control :id
                      (label "user-id" "user id")
                      (text-field {:tabindex 1} "id" id))
             (control :pass
                      (label "pass" "password")
                      (password-field {:tabindex 2} "pass"))
             (control :pass1
                      (label "pass1" "retype password")
                      (password-field {:tabindex 3} "pass1"))
             (submit-button {:tabindex 4} "create account"))))
```

这张表单现在看起来整洁很多了，并且也提供了我们期待的功能（如图 5-5 所示）。

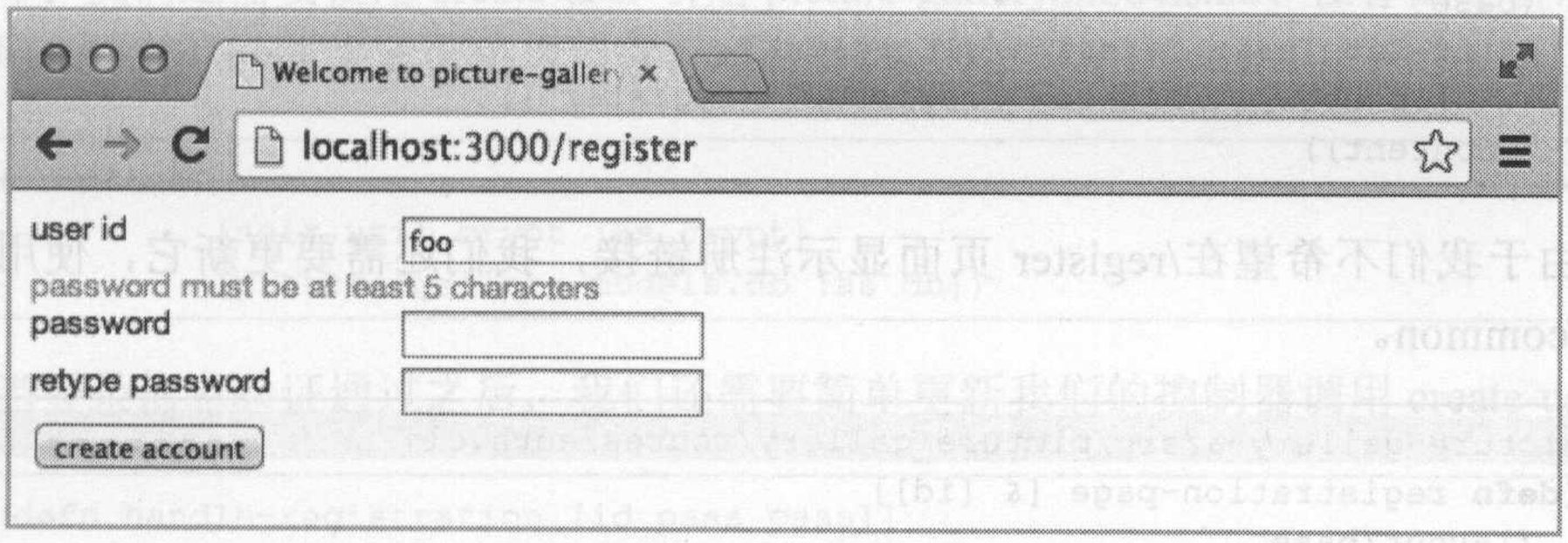

图 5-5 样式化的注册页面

需要格外注意的是，还存在少量代码并没有被简化。使用代码重构来简化也不是绝对的，同时也不能丧失函数的整洁性和可读性，防止过犹不及。

现在再为注册页面添加一个链接，方便用户看见。最合乎逻辑的地方应该是将辅助函数放置在 picture-gallery.views.layout 命名空间下。这可以让所有页面都能使用此布局来显示注册链接。我们打开 picture-gallery.views.layout 命名空间并在定义部分添加一些额外的库引用。

picture-gallery-a/src/picture_gallery/views/layout.clj

```
(ns picture-gallery.views.layout
  (:require [hiccup.page :refer [html5 include-css]]
            [hiccup.element :refer [link-to]]
            [noir.session :as session]))
```

我们需要 noir.session 去检查用户存在与否，还需要从 hiccup.element 取出 link-to 函数用来提供到注册页面的链接。由于并不希望注册页面本身显示此链接，我们要重命名当前版本 common 为 base。

picture-gallery-a/src/picture_gallery/views/layout.clj

```
(defn base [& content]
  (html5
    [:head
     [:title "Welcome to picture-gallery"]
     (include-css "/css/screen.css")]
    [:body content]))
```

现在，我们可以添加一个新的 common 函数，用来检查会话里是否存在用户。如果有，这一布局将显示 ID；如果没有，则显示注册页面的链接。

picture-gallery-a/src/picture_gallery/views/layout.clj

```
(defn common [& content]
  (base
    (if-let [user (session/get :user)]
      [:p user] (link-to "/register" "register"))
    content))
```

由于我们不希望在/register 页面显示注册链接，我们还需要更新它，使用 base 取代 common。

picture-gallery-a/src/picture_gallery/routes/auth.clj

```
(defn registration-page [& [id]]
  (layout/base
    (form-to [:post "/register"]
             (control :id
             (label "user-id" "user id")
             (text-field {:tabindex 1} "id" id))
           (control :pass
             (label "pass" "password")
             (password-field {:tabindex 2} "pass"))
           (control :pass1
             (label "pass1" "retype password")
             (password-field {:tabindex 3} "pass1"))
           (submit-button {:tabindex 4} "create account"))))
```

我们可以试试看，当我们注册一个账户之后并定向到首页，就可以看见显示注册的 ID。我们重启服务，刷新首页，就能看到注册链接。

将用户写入数据库

到目前为止，我们已经将用户保存在会话中。现在，是时候将用户信息储存到数据库中了。我们之前创建过一张用户表，现在只需要创建一个函数将用户详细信息写到这张表即可。让我们再次打开 db 命名空间，并添加如下代码：

picture-gallery-a/src/picture_gallery/models/db.clj

```
(defn create-user [user]
  (sql/with-connection
   db
    (sql/insert-record :users user)))
```

这里的 create-user 接受一个 map 包含两个键。这两个键对应我们数据表的列名，以及我们在网页中创建的表单。函数会简单将数据插入到用户表，如果条目创建失败，会抛出异常。

接下来，我们更新 picture-gallery.routes.auth 命名空间，在账户创建时将用户写入数据库。我们还需要使用 create-user 引用 picture-gallery.models.db。在存储之前，还需要使用 noir.util.crypt 去对密码内容进行哈希处理。

```
(:require ...
          [noir.util.crypt :as crypt]
          [picture-gallery.models.db :as db])
```

当登录信息验证通过之后，我们还需要简单更新我们的控制器调用 create-user。

```
(defn handle-registration [id pass pass1]
  (if (valid? id pass pass1)
    (do
      (db/create-user {:id id :pass (crypt/encrypt pass)})
      (session/put! :user id)
      (resp/redirect "/"))
    (registration-page id)))
```

当试图再次注册用户时，我们应该看见一个新用户记录出现在用户表。我们可以打印数据库内容进行检查，在注册一个用户后，通过在 REPL 中 db 命名空间下运行以下命令：

```
(sql/with-connection db
  (sql/with-query-results res ["select * from users"] (println res)))

({:id foo,
  :pass $2a$10$YZ/9wi7GFmp1wKWShddqFuLUeSyy2rTIEptw2aI9o96TKu5OLqToy})
```

你可能注意到我们的注册页面遇到一点点小意外。如果用户打算注册数据库中已有的 ID，会在页面上提示错误。这不仅是丑陋，还会构成安全风险，是我们程序内部的潜在威胁。

当用户 ID 被占用，我们需要捕获这个异常，并显示明确的信息。我们回到 picture-gallery.routes.auth 命名空间，修改处理错误实现，并将其显示出来。

现在的页面控制器封装在 try/catch 中以调用 create-user，并在显示/register 页面出现错误时捕获异常，而不再重定向到主页。

此前，我们也曾创建了一个辅助函数去检查错误类型，并提供一个友好的信息，而不是原始异常信息。

picture-gallery-a/src/picture_gallery/routes/auth.clj

```
(defn format-error [id ex]
  (cond
    (and (instance? org.postgresql.util.PSQLException ex)
      (= 0 (.getErrorCode ex)))
    (str "The user with id " id " already exists!")

    :else
    "An error has occured while processing the request"))

(defn handle-registration [id pass pass1]
  (if (valid? id pass pass1)
    (try
      (db/create-user {:id id :pass (crypt/encrypt pass)})
      (session/put! :user id)
      (resp/redirect "/")
      (catch Exception ex
        (vali/rule false [:id (format-error id ex)])
        (registration-page)))
    (registration-page id)))
```

现在，若你再用同样的 ID 去注册用户，会在浏览器页面看到明确的错误指示（如图 5-6 所示）。

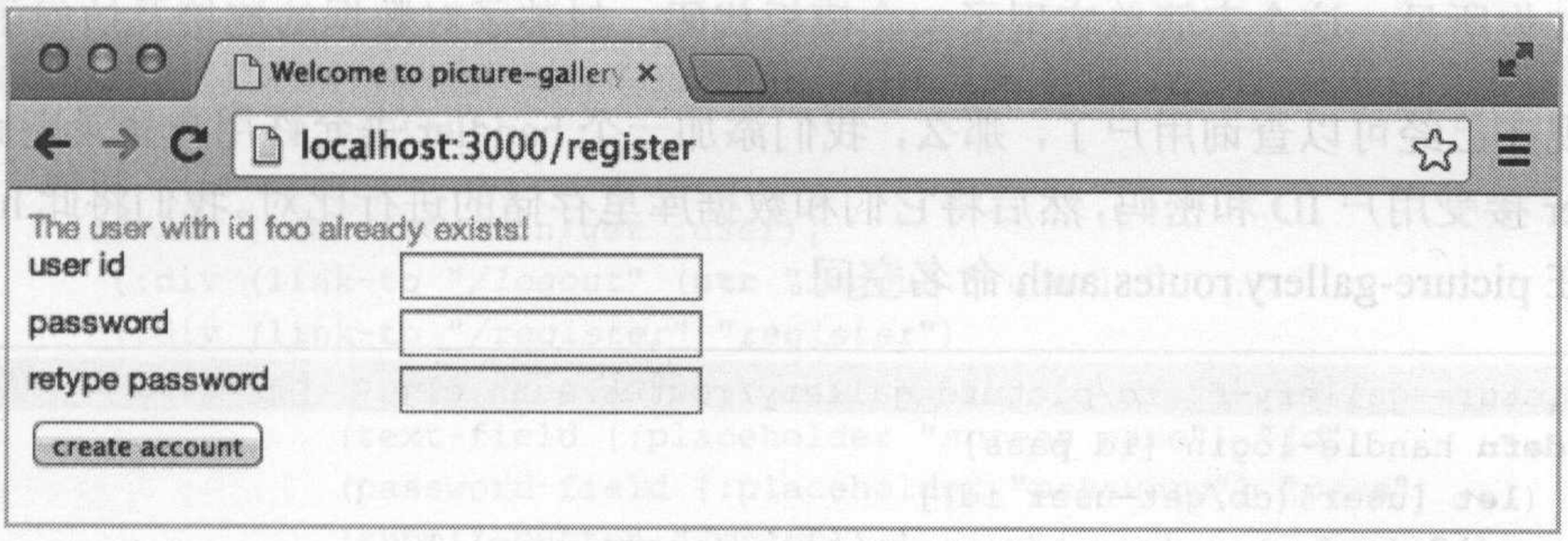

图 5-6 用户名冲突

我们现在已经准备好进行下一个任务：当用户在我们的站点创建账户之后，允许其登录和注销。

5.6 任务 2：登入登出

登入登出操作实现起来很简单，只要用户提交登录表单的 ID 及密码即可。我们只需对照储存的许可，通过则将用户 ID 存入会话。

我们打开 picture-gallery.models.db 命名空间。我们已经添加了储存用户到数据库的函数，现在还需添加一个读取函数，通过给定的 ID 获取用户信息。我们使用 with-query-results 宏获取记录，并返回第一个项。如果没有与给定 ID 匹配的记录，我们简单返回 nil。

```
(defn get-user [id]
  (sql/with-connection
    db
    (sql/with-query-results
      res ["select * from users where id = ?" id] (first res))))
```

由于在每个 db 声明处都需要像(sql/with-connection db ...)这么写，看起来比较重复，我们可以为此写一个简短的宏。

picture-gallery-b/src/picture_gallery/models/db.clj

```
(defmacro with-db [f & body]
  `(sql/with-connection ~db (~f ~@body)))

(defn create-user [user]
  (with-db sql/insert-record :users user))

(defn get-user [id]
  (with-db sql/with-query-results
    res ["select * from users where id = ?" id] (first res)))
```

如你所见，这个宏简单实现了一个模板代码，封装了对数据的连接及访问函数。

现在已经可以查询用户了，那么，我们添加一个 handler 去允许用户登录吧。这个 handler 接受用户 ID 和密码，然后将它们和数据库里存储的进行比对。我们将此 handler 放置在 picture-gallery.routes.auth 命名空间。

picture-gallery-b/src/picture_gallery/routes/auth.clj

```
(defn handle-login [id pass]
  (let [user (db/get-user id)]
    (if (and user (crypt/compare pass (:pass user)))
      (session/put! :user id)))

  (resp/redirect"/"))
```

当用户注销时，我们应该去清理会话，注销 handler 则是简单调用 session/clear!去移除在会话中积累的所有用户数据。

```
(defn handle-logout []
  (session/clear!)
  (resp/redirect "/"))
```

为了将功能暴露给客户端，我们现在需要在定义的 auth-routes 中添加控制路由。

picture-gallery-b/src/picture_gallery/routes/auth.clj

```
(defroutes auth-routes
  (GET "/register" []
       (registration-page))

  (POST "/register" [id pass pass1]
       (handle-registration id pass pass1))

  (POST "/login" [id pass]
       (handle-login id pass))

  (GET "/logout" []
       (handle-logout)))
```

我们还需要为 handler 添加内容以处理来自用户提交的表单。由于这些表单应用于所有页面，由布局来处理将是很好的选择。

我们还需稍作调整来提供登入登出链接，并在命名空间声明处加载 hiccup.form。

```
(:require ... [hiccup.form :refer :all])
```

接下来就好办了，我们更新 common 布局添加登录表单并为注销提供链接。

picture-gallery-b/src/picture_gallery/views/layout.clj

```
(defn common [& content]
  (base
    (if-let [user (session/get :user)]
      [:div (link-to "/logout" (str "logout " user))]
      [:div (link-to "/register" "register")
       (form-to [:post "/login"]
                (text-field {:placeholder "screen name"} "id")
                (password-field {:placeholder "password"} "pass")
                (submit-button "login"))])
    content))
```

我们现在应该能够通过注册的账户来测试登入登出功能。我们已经完成实现所有的用户验证任务，接下来，应该将注意力移至下一个任务：让用户上传图片。

5.7 任务 3：上传图片

因为创建了一个新的工作流，我们应该为其创建一个新的命名空间。我们来构造一个名为 picture-gallery.routes.upload 的命名空间去处理这个任务。

这个工作流请求一种形式方便上传。一旦一个文件通过这种方式提交，当展开图片清单的时候，我们就需要去创建一个缩略图来显示。

由 Java 标准库提供的 java.awt.geom 的包，可以实现图像缩放功能。在这一章，我们将在程序中展示如何通过地道的 Clojure 函数式语句封装 Java 调用。

我们首先在命名空间中导入用到的引用。虽然看着有点多，不过有些应该很熟悉了，我们还会讲到如何将其简短化。

picture-gallery-c/src/picture_gallery/routes/upload.clj

```
(ns picture-gallery.routes.upload
  (:require [compojure.core :refer [defroutes GET POST]]
            [hiccup.form :refer :all]
            [hiccup.element :refer [image]]
            [hiccup.util :refer [url-encode]]
            [picture-gallery.views.layout :as layout]
            [noir.io :refer [upload-file resource-path]]
            [noir.session :as session]
            [noir.response :as resp]
            [noir.util.route :refer [restricted]]
            [clojure.java.io :as io]
            [ring.util.response :refer [file-response]])
```

```
            [picture-gallery.models.db :as db]
            [picture-gallery.util :refer [galleries gallery-path]])

  (:import [java.io File FileInputStream FileOutputStream]
           [java.awt.image AffineTransformOp BufferedImage]
           java.awt.RenderingHints
           java.awt.geom.AffineTransform
           javax.imageio.ImageIO))
```

下一步，我们需要创建一个函数去显示上传页面，以及一个 handler 去处理某种形式的 POST 操作。页面须为 multipart/form-data 类型。handler 将打印我们的参数并显示页面，这将让我们知道提交的内容。

picture-gallery-c/src/picture_gallery/routes/upload.clj

```
(defn upload-page [info]
  (layout/common
    [:h2 "Upload an image"]
    [:p info]
    (form-to {:enctype "multipart/form-data"}
             [:post "/upload"]
             (file-upload :file)
             (submit-button "upload"))))

(defn handle-upload [params]
  (println params)
  (upload-page "success"))
```

通常，我们还得更新路由定义，并在 picture-gallery.handler 命名空间下的 app 声明中添加创建的路由。

```
(defroutes upload-routes
  (GET "/upload" [info] (upload-page info))

  (POST "/upload" {params :params} (handle-upload params)))
```

在 picture-gallery.handler 命名空间下的 app 应该如下声明：

```
(:require ... [picture-gallery.routes.upload :refer [upload-routes]])
...
(def app (noir-middleware/app-handler
          [auth-routes
           home-routes
           upload-routes
           app-routes]))
```

我们可以在浏览器测试这个表达式，可以看见已经成功调用了。当调用之后，handler 应该在控制台打印参数。

```
{:file
  {:size 15,
   :tempfile #<File /var/folders/mv/sch8x99yc30gp/T/ring-multipart-60387396.tmp>,
   :content-type application/octet-stream,
   :filename cloud.jpg}}
```

我们可以看见参数有一个键名为:file，这个:file 键指向一个 map，其包含以下键值：

- :tempfile ——文件本身。
- :filename ——上传的文件名。
- :content-type ——上传文件的负载类型。
- :size ——文件的比特大小。

这为我们保存文件提供了所有必要的信息。我们使用:file 参数来更新 handler 及路由定义，并检查文件是否被选择：

```
(defn handle-upload [{:keys [filename] :as file}]
  (println file)
  (upload-page
    (if (empty? filename)
      "please select a file to upload"
      "success")))
(defroutes upload-routes
  (GET "/upload" [info] (upload-page info))

  (POST "/upload" [file] (handle-upload file)))
```

剩下的工作就是将文件保存到磁盘上。程序的 public 文件夹明显是用于存储文件的。不幸的是，当我们将程序打成 JAR 包之后，这个相对路径就没法用了。我们需要创建一个专用文件夹，用于为相册程序存放文件。

我们为其编写一个名为 gallery-path 的辅助函数：

```
(defn gallery-path []
  "galleries")
```

我们使用 noir.io/upload-file 辅助函数去处理文件上传。它接受一条路径和一份 map 来表示文件，以及可选标识表达如果此路径不存在是否要创建。

我们通过控制器简单调用此函数，并捕获当文件不可保存时抛出的异常。异常信息会被作为信息参数显示给用户。如果上传成功，我们会显示图片上传成功。

```
(defn handle-upload [{:keys [filename] :as file}]
  (upload-page
    (if (empty? filename)
      "please select a file to upload"
      (try
        (noir.io/upload-file (gallery-path) file :create-path? true)
        (image {:height "150px"}
          (str "/img/" (url-encode filename)))

        (catch Exception ex
          (str "error uploading file " (.getMessage ex)))))))
```

我们对文件名使用 url-encode 处理，以确保即使包含 URL 字符集不支持的特殊字符也能正确显示。为了显示图像，我们还需要创建一条新的路由和 handler 来维护上传的文件，最终服务于客户端。

```
(defn serve-file [file-name]
  (file-response (str (gallery-path) File/separator file-name)))

(defroutes upload-routes
  ...
  (GET "/img/:file-name" [file-name] (serve-file file-name)))
```

通过上传页面，上传一张图片来测试上传流程是否符合预期。如果一切顺利，我们会看见页面上显示的图片，如图 5-7、图 5-8 所示。

我们现在打开项目根目录的 galleries 文件夹，就能看到创建于此的这个文件。

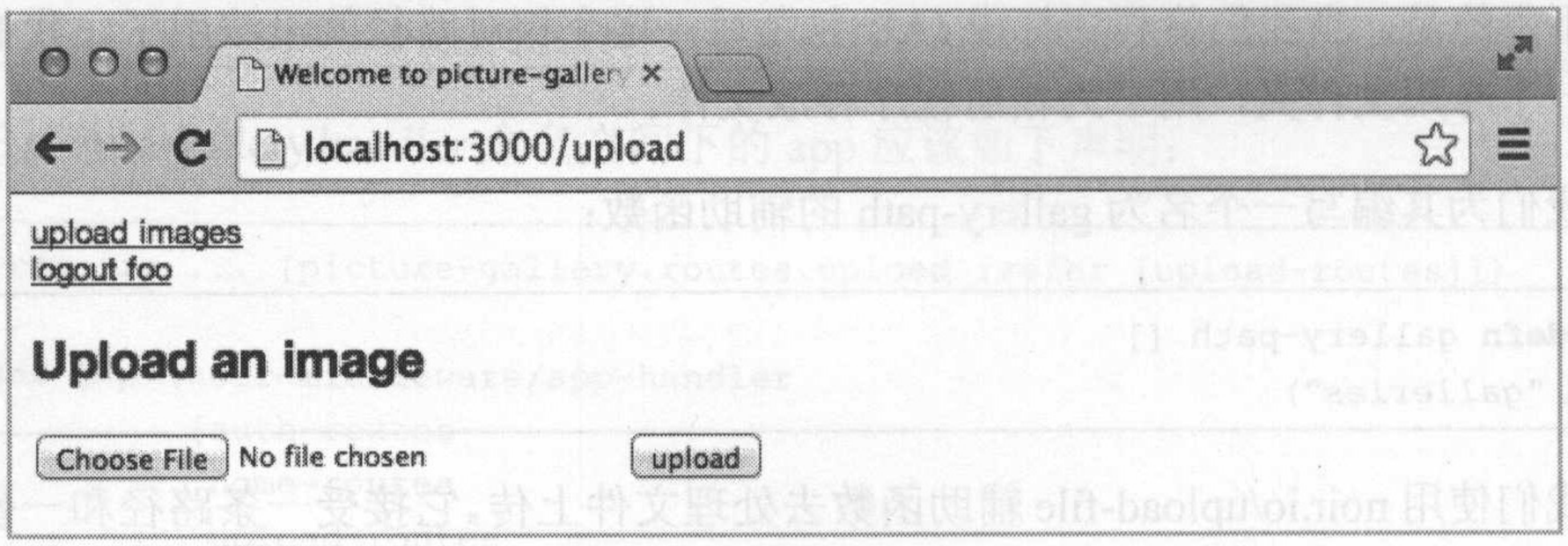

图 5-7　选择上传文件

图 5-8 文件上传成功

生成缩略图

现在，我们已经可以上传文件了，接下来看看在上传文件之后如何生成缩略图。图像需要被缩小并存于一个新文件。我们定义缩略图的尺寸为 150×150 像素，并为缩略图文件名添加 thumb_的前缀。我们为其定义两个常量。

```
(def thumb-size 150)
(def thumb-prefix "thumb_")
```

接下来，我们需要编写一个函数用来缩放图像。这里我们利用由 Java 标准库提供的 AffineTransform 类创建缩放操作对象，使用 AffineTransformOp 去执行变换。transform-op 的 filter 方法会使用我们提供的原始图像去生成缩放后的图像。

picture-gallery-c/src/picture_gallery/routes/upload.clj

```
(defn scale [img ratio width height]
  (let [scale         (AffineTransform/getScaleInstance
                       (double ratio) (double ratio))

        transform-op (AffineTransformOp.
                      scale AffineTransformOp/TYPE_BILINEAR)]
    (.filter transform-op img (BufferedImage. width height (.getType img)))))
```

那我们来试试缩放函数是否已经能正常工作，复制一份图片文件到我们的工程根目录，并将其命名为 image.jpg，接下来在 REPL 中运行如下代码：

```
(ImageIO/write
  (scale (ImageIO/read (io/input-stream "image.jpg")) 0.5 150 150)
  "jpeg"
  (File. "scaled.jpg"))
```

如果函数运行正确，我们应该在同样的文件夹看到 scaled.jpg 文件，图像尺寸为 150×150 像素。

接下来，我们使用 ImageIO 类来读取上传的图片文件。一旦确认图片，我们就获取其长宽尺寸，并将其按照由 thumb-size 常量定义的尺寸来缩放。

picture-gallery-c/src/picture_gallery/routes/upload.clj

```
(defn scale-image [file]
  (let [img          (ImageIO/read file)
        img-width    (.getWidth img)
        img-height   (.getHeight img)
        ratio        (/ thumb-size img-height)]
    (scale img ratio (int (* img-width ratio)) thumb-size)))
```

我们同样要在 REPL 中调用 scale-image 来测试其效果。我们最好提供几个不同尺寸的图片来测试，检验所有的图片都能被正确缩放。

```
(ImageIO/write
  (scale-image (io/input-stream "image.jpg"))
  "jpeg"
  (File. "scaled.jpg"))
```

现在，还剩下写个函数去保存缩略图，用在控制器调用 upload-file 后调用。

picture-gallery-c/src/picture_gallery/routes/upload.clj

```
(defn save-thumbnail [{:keys [filename]}]
  (let [path (str (gallery-path) File/separator)]
    (ImageIO/write
    (scale-image (io/input-stream (str path filename)))
    "jpeg"
    (File. (str path thumb-prefix filename)))))
```

在 save-thumbnail 函数中，我们调用 resource-path 去获取 public 文件夹路径。这时，我们便可以调用 ImageIO/write 保存 scale-image 的输出建立缩略图文件。我们再对 handle-upload 函数更新，在完成上传之后调用 save-thumbnail。

```
(defn handle-upload [{:keys [filename] :as file}]
  (upload-page
    (if (empty? filename)
"please select a file to upload"
 (try
  ;;save the file and create the thumbnail
  (noir.io/upload-file (gallery-path) file :create-path? true)
  (save-thumbnail file)
  ;;display the thumbnail
  (image {:height "150px"}
    (str "/img/" thumb-prefix (url-encode filename)))
  (catch Exception ex
    (str "error uploading file " (.getMessage ex))))))))
```

现在，如果我们通过上传页面上传一个文件，就可以在 galleries 文件夹中看见此文件和缩略图文件。

将文件保存到用户目录

由于我们的网站并不是只有一个用户，而且每个用户都拥有自己的相册。我们现在需要为每个用户提供一些策略去应对一个独立的相册。最简单的办法就是使用用户注册时的账户 ID，因为它正好就是唯一的。

我们更新一下 gallery-path 函数，在相册文件夹下生成唯一路径，其名称基于当前会话的用户 ID。我们还应基于此路径提取一个独立的变量，当别的用户在浏览的时候就可以使用了。

```
(def galleries "galleries")
(defn gallery-path []
  (str galleries File/separator (session/get :user)))
```

下一步，更新 serve-file 函数及其路由，在查找文件时使用用户 ID。

```
picture-gallery-c/src/picture_gallery/routes/upload.clj
(defn serve-file [user-id file-name]
```

```
    (file-response (str galleries File/separator user-id File/separator file-name)))
```

picture-gallery-c/src/picture_gallery/routes/upload.clj

```
(GET "/img/:user-id/:file-name" [user-id file-name]
     (serve-file user-id file-name))
```

现在，我们是在每次准备保存文件时对路径做存在校验。其实放在注册时创建要更简单、可靠。那我们建立一个名为 create-gallery-path 的新函数，在用户账户创建时调用。

我们通过 gallery-path 函数得到路径，并通过 java.io.File 为用户去创建一个新的目录。我们还需要更新命名空间定义描述依赖以下内容：

```
(ns picture-gallery.routes.auth
  (:require ...
            [picture-gallery.routes.upload :refer [gallery-path]])
  (:import java.io.File))
```

接下来，我们添加 create-gallery-path 函数供 handle-registration 调用，确保每次在用户注册时创建一个新的相册。

picture-gallery-c/src/picture_gallery/routes/auth.clj

```
(defn create-gallery-path []
  (let [user-path (File. (gallery-path))]
    (if-not (.exists user-path) (.mkdirs user-path))
    (str (.getAbsolutePath user-path) File/separator)))
```

picture-gallery-c/src/picture_gallery/routes/auth.clj

```
(defn handle-registration [id pass pass1]
  (if (valid? id pass pass1)
    (try
      (db/create-user {:id id :pass (crypt/encrypt pass)})
      (session/put! :user id)
      (create-gallery-path)
      (resp/redirect "/")
      (catch Exception ex
        (vali/rule false [:id (format-error id ex)])
        (registration-page)))
    (registration-page id)))
```

如果我们现在上传一个文件，将在 galleries/<userid>路径下创建此文件和对应的缩略图文件。

你可能注意到我们这里还有点小问题，file-upload 页面并没有涉及用户登录验证。

我们须解决这个问题，方法是在页面显示之前检查会话中的用户。

```
(defn upload-page [info]
  (if (session/get :user)
    (layout/common
      [:h2 "Upload an image"]
      [:p info]
      (form-to {:enctype "multipart/form-data"}
               [:post "/upload"]
               (file-upload :file)
               (submit-button "upload")))
    (resp/redirect "/")))
```

修改的具体方式：当用户尝试在浏览器中打开上传页面时，如果发现会话中并无登录的账户，则将页面重定向到主页。

我们已经处理过这个细节。尽管如此，我们还是需要为上传 handler 再做一遍，包括其他的用户相关页面，也要同样处理。每次都要这样写下判断是一件单调且易疏漏的事情。

好在 lib-noir 提供一种途径去定义规则来限制账户访问页面。那我们来看看怎么创建规则，在页面显示之前去判断给定的用户。

我们切换到 picture-gallery.handler 命名空间，创建一个名为 user-page 的新函数。这个函数必须接受一个请求 map 作为参数，且用于判定一个 URI 能否被访问。我们的案例中，在允许用户访问受限页面之前，我们希望知道会话中的账户。

```
(ns picture-gallery.handler
  ...
  (:require ... [noir.session :as session]))
```

picture-gallery-c/src/picture_gallery/handler.clj

```
(defn user-page [_]
  (session/get :user))
```

user-page 参数列中的下划线（_）简单说明此参数会被忽略。

我们现在需要更新我们的 handler 通过使用:access-rules 键去设置访问规则。app-handler 需要使用 noir.util.middleware/wrap-access-rules 中间件对受限页面应用访问规则。在我们的案例中，我们有个简单规则——user-page 函数。

picture-gallery-c/src/picture_gallery/handler.clj

```
(def app (noir-middleware/app-handler
          [auth-routes
           home-routes
           upload-routes
           app-routes]
           :access-rules [user-page]))
```

在访问规则的地方，我们可以使用 noir.util.route/restricted 宏去对页面做限制。我们来使用这个宏更新上传路由吧。

```
(ns picture-gallery.routes.upload
  (:require ... [noir.util.route :refer [restricted]])
 ...)
```

picture-gallery-c/src/picture_gallery/routes/upload.clj

```
(defroutes upload-routes
  (GET "/img/:user-id/:file-name" [user-id file-name]
       (serve-file user-id file-name))
  (GET "/upload" [info] (restricted (upload-page info)))

  (POST "/upload" [file] (restricted (handle-upload file))))
```

我们现在可以移除上传页面的检测代码了，然后测试一下功能是否不变。

picture-gallery-c/src/picture_gallery/routes/upload.clj

```
(defn upload-page [info]
  (layout/common
    [:h2 "Upload an image"]
    [:p info]
    (form-to {:enctype "multipart/form-data"}
             [:post "/upload"]
             (file-upload :file)
             (submit-button "upload"))))
```

使用 restricted 宏比通过 if 判断使程序表述更清晰，意图更明确，而且当规则不再是一条判断而是更加复杂组合的时候，还能灵活调整。

在数据库保存图片清单

由于后面的任务都是围绕图片显示的，我们需要为任务（比如查找具体用户的所有图片）创建一些元数据。

用数据库存放这些元数据是最佳选择。我们需要创建一张表，用来在每次图片上传后保存图片的引用。

我们创建一张名为 images 的表，包含用户 ID 及图片名。接下来我们打开 gallery.models.schema 命名空间，添加一张新表的定义。

picture-gallery-c/src/picture_gallery/models/schema.clj

```
(defn create-images-table []
  (sql/with-connection db
    (sql/create-table
      :images
      [:userid "varchar(32)"]
      [:name "varchar(100)"])))
```

我们现在可以在 REPL 中运行(create-images-table)对此求值，在 images 表就位之后，我们需要在每次上传图片时简单插入一条记录，具体方法是在 db 命名空间下编写一条函数。

picture-gallery-c/src/picture_gallery/models/db.clj

```
(defn add-image [userid name]
  (with-db
    sql/transaction
    (if (sql/with-query-results
          res
          ["select userid from images where userid = ? and name = ?" userid name]
          (empty? res))
      (sql/insert-record :images {:userid userid :name name})
      (throw
       (Exception. "you have already uploaded an image with the same name")))))
```

在 add-image 函数中，我们会检查并判断用户是否已经上传过一张同名图片，以防之前的图片被意外覆盖。

我们现在还需要在 upload 命名空间下添加 db 命名空间引用，在每次上传图片后，在上传图片的控制器中调用 add-image 函数。

```
(ns picture-gallery.routes.upload
  ...
  (:require ... [picture-gallery.models.db :as db]))
```

picture-gallery-c/src/picture_gallery/routes/upload.clj

```
(defn handle-upload [{:keys [filename] :as file}]
  (upload-page
    (if (empty? filename)
```

```
        "please select a file to upload"
        (try
          (upload-file (gallery-path) file)
          (save-thumbnail file)
          (db/add-image (session/get :user) filename)
        (image {:height "150px"}
          (str "/img/"
               (session/get :user)
               "/"
               thumb-prefix
              (url-encode filename)))
        (catch Exception ex
           (str "error uploading file " (.getMessage ex))))))))
```

下一次上传图片的时候，我们可以在数据库中检查一下，应该有一条新记录。我们现已完成所有关于用户上传文件的功能。

我们需要花点时间去消化上述内容。

重构通用代码

我们已编写的一些代码也可用于其他页面。比如，当我们要显示照片的时候，我们需要知道缩略图前缀名以及 gallery 路径。

为应对接下来的任务，我们新增一个 picture-gallery.util 命名空间，并将 thumb-prefix、galleries、gallery-path 这三个函数放置于此。除此之外，还要创建几个用来生成图片和缩略图 URI 的函数。

picture-gallery-c/src/picture_gallery/util.clj

```
(ns picture-gallery.util
  (:require [noir.session :as session]
            [hiccup.util :refer [url-encode]])
  (:import java.io.File))

(def thumb-prefix "thumb_")
(def galleries "galleries")

(defn gallery-path []
  (str galleries File/separator (session/get :user)))

(defn image-uri [userid file-name]
  (str "/img/" userid "/" (url-encode file-name)))

(defn thumb-uri [userid file-name]
  (image-uri userid (str thumb-prefix file-name)))
```

我们现在还得调整 auth 和 upload 命名空间依赖 util 命名空间。

```
(ns picture-gallery.routes.auth
  (:require ...
            [picture-gallery.util
             :refer [gallery-path]]))
(ns picture-gallery.routes.upload
  (:require ...
            [picture-gallery.util
             :refer [galleries gallery-path thumb-prefix thumb-uri]]))
```

现在可以通过 thumb-uri 辅助函数调整以下用来处理上传的代码：

```
picture-gallery-d/src/picture_gallery/routes/upload.clj
(defn handle-upload [{:keys [filename] :as file}]
  (upload-page
    (if (empty? filename)
      "please select a file to upload"
      (try
        (upload-file (gallery-path) file)
        (save-thumbnail file)
        (db/add-image (session/get :user) filename)
        (image {:height "150px"}
          (thumb-uri (session/get :user) filename))
        (catch Exception ex
          (str "error uploading file " (.getMessage ex)))))))
```

现在，我们对上传代码进行了重构，当会话存在用户，我们必须在通用布局中提供一个上传链接。

```
(defn common [& content]
  (base
    (if-let [user (session/get :user)]
      (list
        [:div (link-to "/upload" "upload images")]
        [:div (link-to "/logout" (str "logout " user))])
      [:div (link-to "/register" "register")
       (form-to [:post "/login"]
                (text-field {:placeholder "screen name"} "id")
                (password-field {:placeholder "password"} "pass")
                (submit-button "login"))])
    content))
```

我们做了关于会话中存在用户的检测，并显示一个注销按钮。当我们完成几个页

面后，我们的用户菜单也会随之增长，这就意味着我们还需要将此功能独立出来。

picture-gallery-c/src/picture_gallery/views/layout.clj

```
(defn guest-menu []
  [:div (link-to "/register" "register")
   (form-to [:post "/login"]
            (text-field {:placeholder "screen name"} "id")
            (password-field {:placeholder "password"} "pass")
            (submit-button "login"))])

(defn user-menu [user]
  (list
    [:div (link-to "/upload" "upload images")]
    [:div (link-to "/logout" (str "logout " user))]))

(defn common [& content]
  (base
    (if-let [user (session/get :user)]
      (user-menu user)
      (guest-menu))
    content))
```

我们先切换到主页，然后测一下在重构之后所有功能是否还能符合预期。特别应该像之前一样浏览一下上传页面、登录页面以及上传的文件。

5.8　任务4：显示图片

我们现在已经具备了所有相册的基础，可以开始处理显示图片了。我们将与用户相关的缩略图简单加载，并显示在页面上。当用户点击一张缩略图时，会显示为全尺寸的图片。

由于上传的图片引用存放在数据库，我们可以简单编写一个函数，通过给定的用户ID来查询所有相关的图片。

picture-gallery-d/src/picture_gallery/models/db.clj

```
(defn images-by-user [userid]
  (with-db
    sql/with-query-results
    res ["select * from images where userid = ?" userid] (doall res)))
```

我们接下来测试这个函数，看它能不能通过给定一个之前注册的账户来获取这个用户上传的图片清单。

```
(images-by-user "foo")

({:name "logo.jpg", :userid "foo"})
```

我们可以在用户登录之后的欢迎页面中显示缩略图。这将是一条全新的工作流，我们创建一个名为 picture-gallery.routes.gallery 的新命名空间，并添加围绕显示用户相册的功能。

同样，我们在命名空间声明处添加所有需要的依赖项，接下来，再看看怎么将它们用于新功能中。

```
picture-gallery-d/src/picture_gallery/routes/gallery.clj
(ns picture-gallery.routes.gallery
  (:require [compojure.core :refer :all]
            [hiccup.element :refer :all]

            [picture-gallery.views.layout :as layout]
            [picture-gallery.util
             :refer [thumb-prefix image-uri thumb-uri]]

            [picture-gallery.models.db :as db]
            [noir.session :as session]))
(defn thumbnail-link [{:keys [userid name]}]
  [:div.thumbnail
   [:a {:href (image-uri userid name)}
    (image (thumb-uri userid name))]])
(defn display-gallery [userid]
  (or
   (not-empty (map thumbnail-link (db/images-by-user userid)))
   [:p "The user " userid " does not have any galleries"]))

(defn gallery-link [{:keys [userid name]}]
  [:div.thumbnail
   [:a {:href (str "/gallery/" userid)}
    (image (thumb-uri userid name))
    userid "'s gallery"]])

(defn show-galleries []
    (map gallery-link (db/get-gallery-previews)))

(defroutes gallery-routes
  (GET "/gallery/:userid" [userid] (layout/common (display-gallery userid))))
```

我们创建两个辅助函数：第一个是用来生成包含缩略图链接的 div 标签，并添加 thumbnail 类；第二个是通过会话中的用户读取图片，再通过 thumbnail-link 函数将这些图

片转换为缩略图。对于用户还没有图片这种情况，我们提供一个辅助信息传达给用户。

picture-gallery-d/src/picture_gallery/routes/gallery.clj

```
(defn thumbnail-link [{:keys [userid name]}]
  [:div.thumbnail
   [:a {:href (image-uri userid name)}
    (image (thumb-uri userid name))]])
(defn display-gallery [userid]
  (or
    (not-empty (map thumbnail-link (db/images-by-user userid)))
    [:p "The user " userid " does not have any galleries"]))
```

我们同样定义一条新路由为给定的用户 ID 显示相册。

picture-gallery-d/src/picture_gallery/routes/gallery.clj

```
(defroutes gallery-routes
  (GET "/gallery/:userid" [userid] (layout/common (display-gallery userid))))
```

这意味着我们还需要将 picture-gallery.routes.gallery 添加到 picture-gallery.handler 命名空间，并添加一条新路由到路由表。

```
(:require ...
          [picture-gallery.routes.gallery :refer [gallery-routes]])
...
(def app (noir-middleware/app-handler
           [auth-routes
            home-routes
            upload-routes
            gallery-routes
            app-routes]
            :access-rules [user-page]))
```

picture-gallery-d/src/picture_gallery/handler.clj

```
(def app (noir-middleware/app-handler
           [auth-routes
            home-routes
            upload-routes
            gallery-routes
            app-routes]
           :access-rules [user-page]))
```

如果我们创建一个名为 foo 的用户并上传了一些图片，我们打开 http://localhost:3000/gallery/foo 就能看见我们上传的这些图片。如果我们在此键入一个非法的用户 ID，可以顺便测试一下能不能得到一个友好提示。

我们还需要更新位于 resources/public/css 路径下的 screen.css，为缩略图添加一点样式化的处理。

picture-gallery-d/resources/public/css/screen.css

```
.thumbnail {
    float:left;
    padding: 5px;
    margin: 8px;
    border-style:solid;
    border-width:1px;
    border-color:#ccc;
    box-shadow:          4px 4px 6px -1px #222;
    -moz-box-shadow:     4px 4px 6px -1px #222;
    -webkit-box-shadow: 4px 4px 6px -1px #222;
}
```

剩下的事情就是在首页上显示一个通往相册的链接。要做到这一点，我们还要添加另外一个函数到 db 命名空间。这个函数用来从用户相册提取一张图片。

picture-gallery-d/src/picture_gallery/models/db.clj

```
(defn get-gallery-previews []
  (with-db
    sql/with-query-results
    res
    ["select * from
      (select *, row_number() over (partition by userid) as row_number from images)
      as rows where row_number = 1"]
    (doall res)))
```

我们还需要在 gallery 命名空间添加一个函数，用于生成相册的链接。这有点类似于 thumbnail-link 函数，只是这个函数是链接到相册页面的。

picture-gallery-d/src/picture_gallery/routes/gallery.clj

```
(defn gallery-link [{:keys [userid name]}]
  [:div.thumbnail
   [:a {:href (str "/gallery/" userid)}
    (image (thumb-uri userid name))
    userid "'s gallery"]])
```

我们现在再创建一个辅助函数，通过给定用户显示所有有效图像。

picture-gallery-d/src/picture_gallery/routes/gallery.clj

```
(defn show-galleries []
    (map gallery-link (db/get-gallery-previews)))
```

最后，我们可以更新首页，通过我们站点的所有用户显示有效相片。

picture-gallery-d/src/picture_gallery/routes/home.clj

```
(ns picture-gallery.routes.home
  (:require [compojure.core :refer :all]
            [picture-gallery.views.layout :as layout]
            [noir.session :as session]
            [picture-gallery.routes.gallery :refer [show-galleries]]))
(defn home []
  (layout/common (show-galleries)))
```

现在，通过注册到网站的这些用户，可以打开不同的相册，我们需要在菜单上添加一个主页按钮。既然如此，我们借此机会完善一下菜单。

picture-gallery-d/src/picture_gallery/views/layout.clj

```
(defn make-menu [& items]
  [:div (for [item items] [:div.menuitem item])])
(defn guest-menu []
  (make-menu
   (link-to "/" "home")
   (link-to "/register" "register")
   (form-to [:post "/login"]
            (text-field {:placeholder "screen name"} "id")
            (password-field {:placeholder "password"} "pass")
            (submit-button "login"))))
(defn user-menu [user]
  (make-menu
    (link-to "/" "home")
    (link-to "/upload" "upload images")
    (link-to "/logout" (str "logout " user))))
```

每一个菜单项都包含在一个 menuitem 类的 div 标签内。我们还会添加一些 CSS 以样式化这些标签，这样就能粘在页面的顶部。

picture-gallery-d/resources/public/css/screen.css

```
.menuitem {
    float: left;
    margin-right: 10px;
}
.content {
    clear: both;
    padding: 20px;
}
```

我们同时也添加了一个 content 类，用来清除左浮动，并放在菜单后面添加菜单内

容。我们通过此类封装一个包含内容的 div。

picture-gallery-d/src/picture_gallery/views/layout.clj

```
(defn common [& content]
  (base
    (if-let [user (session/get :user)]
      (user-menu user)
      (guest-menu))
    [:div.content content]))
```

至此，我们已经打造了一个网站，可以让用户注册账户、登录、上传图片、根据用户分组浏览图片。我们现在再添加一个新功能，用来让用户删除上传的图片。

5.9 任务 5：删除图片

当用户不再想看到某张图片时，希望移除图片也是合情合理的。我们需要为用户提供一个途径，通过此途径用户可以选择他们想删除的图片，并告知要删除图片。我们的处理内容如下：

- 删除图片。
- 删除缩略图。
- 删除图片的数据库条目。

由于图片仅能被所有者删除，我们就需要检查页面和会话中的用户是否匹配。匹配则允许用户标记需要删除的图片，并通过 delete 按钮提交选择。

到目前为止，我们仅创建过静态页。接下来，我们看看如何通过 Ajax 添加一些客户端行为。在本章，我们会在页面引用 JavaScript，通过浏览器使用 HTTP POST 访问服务端处理，并通过 JSON（JavaScript Object Notation，JavaScript 对象表示法）返回响应给客户端。

我们使用 Ajax 通知服务端，告知其需要删除的图片并删除，接下来更新页面并显示操作结果。

首先，我们在 picture-gallery.models.db 命名空间中新建一个函数，用来从数据库删除图片。

picture-gallery-e/src/picture_gallery/models/db.clj

```
(defn delete-image [userid name]
  (with-db
    sql/delete-rows :images ["userid=? and name=?" userid name]))
```

接下来，我们在 picture-gallery.routes.upload 命名空间添加一个函数，用来执行删

除的三个任务。我们需要提供一个函数根据用户 ID 及图片名来完成这些任务。

首先，我们需要将此操作放在 try/catch 中。如果删除成功，返回 ok；如果遇到任何错误，将返回错误消息。

picture-gallery-e/src/picture_gallery/routes/upload.clj

```
(defn delete-image [userid name]
  (try
    (db/delete-image userid name)
    (io/delete-file (str (gallery-path) File/separator name))
    (io/delete-file (str (gallery-path) File/separator thumb-prefix name))
    "ok"
    (catch Exception ex (.getMessage ex))))
```

其次，我们需要添加 handler 去删除多个图片以及处理路由。

picture-gallery-e/src/picture_gallery/routes/upload.clj

```
(defn delete-images [names]
  (let [userid (session/get :user)]
    (resp/json
      (for [name names] {:name name :status (delete-image userid name)}))))

(defroutes upload-routes
  (GET "/img/:user-id/:file-name" [user-id file-name]
       (serve-file user-id file-name))

  (GET "/upload" [info] (restricted (upload-page info)))

  (POST "/upload" [file] (restricted (handle-upload file)))

  (POST "/delete" [names] (restricted (delete-images names))))
```

再次，delete-images 接受要删除的图片名称列表。我们现在从会话中提取用户 ID，将图片名称清单传给 delete-images，并将每一个操作返回给客户端。

由于要通过 Ajax 选择多个缩略图，并调用/delete 路由，我们需要在页面添加一些 JavaScript 代码。我们在 resources/public/js 下新建一个名为 gallery.js 的文件。在我们的相册页面将加载这个文件，届时，我们将为客户端提供管理相册的功能。

我们现在开始编写一个功能，通过选择多个图片并创建 Ajax 请求。我们将使用 jQuery 来简化 JavaScript，在 base 布局引用它。务必在命名空间定义段声明引用 hiccup.page。

```
(:require ...
      [hiccup.page :refer [html5 include-css include-js]])
...

(html5 base [& content]
```

```
  [:head
   [:title "Welcome to picture-gallery"]
   (include-css "/css/screen.css")
   (include-js "//code.jquery.com/jquery-2.0.2.min.js")]
  [:body content])
```

我们为客户端编写删除图片的功能完成最终准备。

```
function deleteImages() {
var selectedInputs = $("input:checked");
var selectedIds = [];
selectedInputs
.each(function() {
            selectedIds.push($(this).attr('id'));
      });
if (selectedIds.length < 1) alert("no images selected");
else
    $.post("/delete",
          {names: selectedIds},
          function(response) {
              var errors = $('<ul>');
             $.each(response, function() {
             if("ok" === this.status) {
                 var element = document.getElementById(this.name);
                  $(element).parent().parent().remove();
             }
             else
                 errors
                 .append($('<li>',
                         {html: "failed to remove " +
                                this.name +
                                ": " +
                                this.status}));
             });
             if (errors.length > 0)
                 $('#error').empty().append(errors);
      },
     "json");
}
```

在以上代码中，我们为每个项提供多选框输入并收集 ID 属性。下一步，我们使用 HTTP POST 请求将这些 ID 传递给服务端的 delete-imageshandler。

服务端将返回一个更新状态列表。当更新成功，状态会被设置为 ok，我们会删除对应的元素。否则，我们会创建基于状态代码的错误消息并显示给用户。

JavaScript 文件需要在页面上引用才能运行。我们可以使用 include-js 去加载，正如之前在布局中针对 jQuery 所做的那样。由于此 JavaScript 是仅针对相册页面的，我

们直接将其添加在路由定义中。

```
    (:require ... [hiccup.page :refer :all])

picture-gallery-e/src/picture_gallery/routes/gallery.clj
    (defroutes gallery-routes
      (GET "/gallery/:userid" [userid]
           (layout/common
             (include-js "/js/gallery.js")
             (display-gallery userid))))
```

我们还需要在显示缩略图处修改几个地方，现在还没将勾选复选框和缩略图关联。我们添加必要的引用到 gallery 命名空间并更新 thumbnail-link：

```
    (:require ...
              [hiccup.form :refer [check-box]])

picture-gallery-e/src/picture_gallery/routes/gallery.clj
    (defn thumbnail-link [{:keys [userid name]}]
      [:div.thumbnail
       [:a {:class name :href (image-uri userid name)}
        (image (thumb-uri userid name))
        (if (= userid (session/get :user)) (check-box name))]])
```

现在，如果用户 ID 与会话中一致，我们就在缩略图 div 的图像后面显示一个复选框，（如图 5-9 所示）。

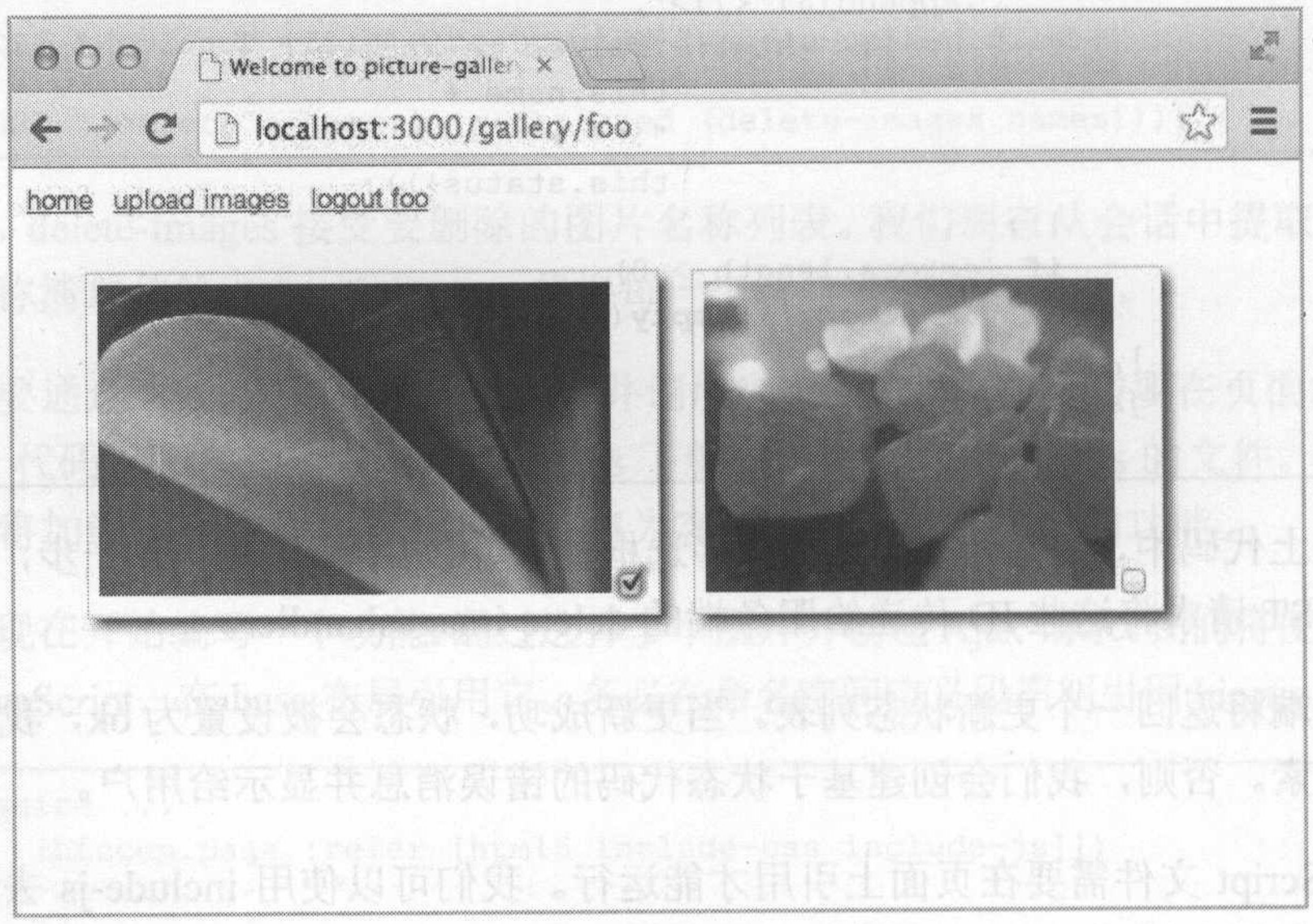

图 5-9　相册复选框

复选框已经就位了，但现在我们还不能直接用。接下来我们还要更新 display-gallery 函数，提供一个 Delete 按钮以及一个用于显示错误信息的 div。

picture-gallery-e/src/picture_gallery/routes/gallery.clj

```
(defn display-gallery [userid]
  (if-let [gallery (not-empty (map thumbnail-link (db/images-by-user userid)))]
    [:div
     [:div#error]
     gallery
     (if (= userid (session/get :user))
       [:input#delete {:type "submit" :value "delete images"}])]
    [:p "The user " userid " does not have any galleries"]))
```

现在，假设我们已打开相册，并且用户就是这个相册的所有者，我们还需要提供一个 Delete 按钮。在页面加载时，我们将这个按钮绑定到 gallery.js 中的 delete 函数。

picture-gallery-e/resources/public/js/gallery.js

```
$(document).ready(function(){
    $("#delete").click(deleteImages);
});
```

我们现在应该可以试试看，当所有者浏览相册时，是不是每个缩略图都有个复选框。如果我们选择了一些图片后按下 Delete 按钮，它们会不会从页面上消失。我们同样可以检查图片、缩略图以及数据库是否被正确删除。

Ajax 和 Servlet Context

在此前的代码中，我们的应用程序在独立运行时一切正常。不过，要是我们将程序放在程序应用服务中运行，Ajax 请求就会出错，这是因为 URL 全地址需应用程序上下文提供前缀。

这对我们来说相当不幸，浏览器并没有意识到我们的应用程序有没有上下文。解决此问题的方式之一是首先在页面中设置一个变量。

请求 map 包含一个名为:context 的键，此键对应的值恰是我们所需要的。看到这里估计你有点犯迷糊了。毕竟，我们并不需要每次都明确地将请求发送给处理端，正是因为我们可以通过上下文提取。

所幸，Compojure 使用 compojure.response.Renderable 约定了 handler 返回 Ring 响

应，这种协议的形式如下：

```
(defprotocol Renderable
  (render [this request]
    "Render the object into a form suitable for the given request map."))
```

如你所见，这个协议定义了一个 render 方法。这个方法接受对象实例以及其后的请求。

使用这个协议前，我们首先需要去添加引用，在 picture-gallery.views.layout 命名空间的定义部分添加 ring.util.response/response。

picture-gallery-e/src/picture_gallery/views/layout.clj

```
(ns picture-gallery.views.layout
  (:require [hiccup.page :refer [html5 include-css]]
            [hiccup.element :refer [link-to]]
            [noir.session :as session]
            [hiccup.form :refer :all]
            [hiccup.page :refer [include-css include-js]]
            [ring.util.response :refer [content-type response]]
            [compojure.response :refer [Renderable]]))
```

为实现协议，我们需要手动设置适当的响应标头，还需要创建 utf-8-response 函数去设置负载类型为 *text/html*，以及编码类型为 *utf-8*。

picture-gallery-e/src/picture_gallery/views/layout.clj

```
(defn utf-8-response [html]
  (content-type (response html) "text/html; charset=utf-8"))
```

下一步，我们创建一个 RenderablePage 类型来扩展 Renderable 协议。我们会将 base 布局的代码转移到 render 操作中。

由于现在可以访问请求了，我们在页面的 head 段落添加 JavaScript 变量，并为其用上下文赋值。

最后，render 方法的代码会封装在我们之前引用的 response 函数中，结果如下：

picture-gallery-e/src/picture_gallery/views/layout.clj

```
(deftype RenderablePage [content]
  Renderable
  (render [this request]
    (utf-8-response
      (html5
        [:head
         [:title "Welcome to picture-gallery"]
```

```
      (include-css "/css/screen.css")
      [:script {:type "text/javascript"}
       (str "var context=\"" (:context request) "\";")]
      (include-js "//code.jquery.com/jquery-2.0.2.min.js")]
     [:body content])))))
```

base 布局函数现在返回 RenderablePage 的实例，而不是直接生成布局：

picture-gallery-e/src/picture_gallery/views/layout.clj

```
(defn base [& content]
  (RenderablePage. content))
```

最终，我们更新 JavaScript，在创建 POST 请求时，使用预置的变量作用于 URL。

picture-gallery-e/resources/public/js/gallery.js

```
$.post(context + "/delete",
       {names: selectedIds},
       function(response) {
          var errors = $('<ul>');
         $.each(response, function() {
          if("ok" === this.status) {
              var element = document.getElementById(this.name);
              $(element).parent().parent().remove();
          }
        else
            errors
            .append($('<li>',
                    {html: "failed to remove " +
                           this.name +
                           ": " +
                           this.status}));
         });
         if (errors.length > 0)
             $('#error').empty().append(errors);
},
"json");
```

现在，上下文就能作用 URL 了。当上下文无效，此变量会包含一个空字符串，请求内容将保持不变。

5.10 任务 6：删除账户

当打算删除用户的账户时，我们需要从数据库删除与用户相关的所有信息，以及用户的所有文件。所幸我们已经写过删除单个图片以及其引用的函数。我们现在要做

的，就是通过给定的用户选择所有相关图片，并将图片一个个传给这个函数。接下来，我们还要删除用户表中的账户，并删除用户目录。

picture-gallery.routes.auth 命名空间包含身份验证逻辑关系，以及管理用户账户。我们打开这个命名空间，添加一个名为 delete-account-page 的函数来控制删除账户逻辑。

很显然，删除账户操作仅针对此时会话中的用户，因此，我们需要对此路由进行限制。首先，我们需要在命名空间定义处引用 noir.util.route。

```
(:require ... [noir.util.route :refer [restricted]])
```

下一步，我们还需要为访问删除账户添加一条路由。

```
picture-gallery-f/src/picture_gallery/routes/auth.clj
(GET "/delete-account" []
  (restricted (delete-account-page)))
```

当用户选择删除账户，我们最好确定此行为不是误操作。account-deletion-page 会重定向到确认页面，给用户选择退出的确认选项。

```
picture-gallery-f/src/picture_gallery/routes/auth.clj
(defn delete-account-page []
  (layout/common
    (form-to [:post "/confirm-delete"]
      (submit-button "delete account"))
    (form-to [:get "/"]
      (submit-button "cancel"))))
```

接下来，我们添加一条用来处理账户删除确认页面的路由。

```
picture-gallery-f/src/picture_gallery/routes/auth.clj
(POST "/confirm-delete" []
      (restricted (handle-confirm-delete)))
```

如果用户发起了/confirm-delete 请求，那我们就知道，用户确实想删除账户。接下来，我们执行必要的任务就行了。

```
picture-gallery-f/src/picture_gallery/routes/auth.clj
(defn handle-confirm-delete []
  (let [user (session/get :user)]
    (doseq [{:keys [name]} (db/images-by-user user)]
      (delete-image user name))
    (clojure.java.io/delete-file (gallery-path))
```

```
    (db/delete-user user))
  (session/clear!)
  (resp/redirect "/"))
```

这些任务包括：调用 upload 命名空间的 delete-image 函数删除用户上传的每一张图片；从数据库的用户表中删除用户；删除用户图片文件夹。完成账户数据删除之后，我们就可以清理会话并将用户重定向到首页。

首先，我们需要在命名空间定义处引用 picture-gallery.routes.upload，这样才能使用 delete-image 函数。

```
(:require ... [picture-gallery.routes.upload :refer [delete-image]])
```

最后，我们需要在 db 命名空间创建一个名为 delete-user 的函数，用来从数据表中删除用户。

```
picture-gallery-f/src/picture_gallery/models/db.clj
(defn delete-user [userid]
  (with-db sql/delete-rows :users ["id=?" userid]))
```

我们现在对以上工作进行测试，试图移除一个在测试中添加的用户。如果一切如预期顺利进行，我们就可以添加一个链接到用户菜单中，将这个功能提供给用户。

```
picture-gallery-f/src/picture_gallery/views/layout.clj
(defn user-menu [user]
  (make-menu
    (link-to "/" "home")
    (link-to "/upload" "upload images")
    (link-to "/logout" (str "logout " user))
    (link-to "/delete-account" "delete account")))
```

这便是在本章一开始我们所提到的一系列的任务，我们现在都已完成，并实现了一个更为完善的多用户相册网站。

5.11 你学到什么

在本章，我们将之前章节所学到的技巧方法，融汇到我们的程序中。在下一章，我们将继续完善程序，准备如何部署程序。

第 6 章

收尾

我们已经创建程序，并完成了我们起初设计的工作流。仅通过这些功能，暂时还没有什么很吸引人的地方。接下来，我们看看如何通过使用 CSS 和 JavaScript 来美化用户界面。

6.1 添加一些样式

我们在第 1 章讨论过，你不应该直接在 Hiccup 模板中嵌入 CSS，而是通过为元素分配适当的 ID 和类标签。这样，我们就可以在 CSS 文件中对指定的元素进行样式配置。

首先，我们为页面的 body 设置一个大体的样式。我们打开 screen.css 进行如下样式设置：

picture-gallery-style-tests/resources/public/css/screen.css

```
body {
    margin: 0px;
    background: #C8D9C9;
    color: #525952;
    font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
    font-size: 13px;
}
```

在页面的 body 中，我们将 margin（边缘）设置为 0 像素，并对 background（背景）、colors（文本色）和 font-family（默认字体）进行了相应设置。

下一步，我们再将菜单设置得美观一些。这里，我们先为菜单的 div 设置 ID，称之为 usermenu，并将每一个菜单项设置为 menuitem 类。

picture-gallery-style-tests/src/picture_gallery/views/layout.clj

```
(defn make-menu [& items]
  [:div#usermenu (for [item items] [:div.menuitem item])])
```

接下来，我们就可以在 CSS 中为菜单标签配置样式了，其宽度横跨整页，底部有阴影。我们还需要将 display 指定为 inline-block（内联块）属性，确保其传递给包含的菜单项。

picture-gallery-style-tests/resources/public/css/screen.css

```
#usermenu {
    background-color: #878C87;
    width: 100%;
    border-bottom-color: #dedede;
    line-height: 25px;
    border:1px solid;
    border-color:#ccc;
    -webkit-box-shadow: 0 8px 6px -6px #555;
    -moz-box-shadow: 0 8px 6px -6px #555;
    box-shadow: 0 8px 6px -6px #555;
    margin-bottom: 25px;
    display: inline-block;
}
```

然后，我们对菜单链接进行样式配置，使其在视觉上凸显出来。

picture-gallery-style-tests/resources/public/css/screen.css

```
#usermenu a:link {text-decoration: none; color: white;}
#usermenu a:hover {text-decoration: underline; color: white;}
#usermenu a:visited {text-decoration: none; color: white;}
```

现在，我们来对菜单项进行样式处理。

picture-gallery-style-tests/resources/public/css/screen.css

```
.menuitem {
    color: #D1F0DA;
    font-size: 15px;
    font-weight:bold;
    float: left;
    list-style: none;
    padding: 5px;
    margin: 5px;
}
```

我们同样可以样式化提交按钮，让其看起来漂亮一点，设置其尺寸和边框。

picture-gallery-style-tests/resources/public/css/screen.css

```
input[type=submit] {
    background: #99A699;
    padding: 5px 8px 5px;
    color: #fff;
    font-weight: bold;
    -moz-border-radius: 5px;
    -webkit-border-radius: 5px;
    -moz-box-shadow: 0 1px 3px #999;
    -webkit-box-shadow: 0 1px 3px #999;
    text-shadow: 0 -1px 1px #222;
    border-bottom: 1px solid #222;
    cursor: pointer;
}
```

通过 JavaScript 添加颜色

我们可以通过 JavaScript 对缩略图添加一些样式。我们使用 AlbumColors 库[①]（请在 GitHub 搜索）来提取图片的色调，并设置缩略图的 div 背景色。

我们将库存放在 resources/public/js 中，命名为 colors.js，并为网站设置一个名为 site.js 的 JavaScript 文件。我们现在在 base 布局引用这些文件。

picture-gallery-style-tests/src/picture_gallery/views/layout.clj

```
(include-js "//code.jquery.com/jquery-2.0.2.min.js"
            "/js/colors.js"
            "/js/site.js")
```

在 site.js 中，我们使用 AlbumColors 提取缩略图颜色并设置对应的 div 风格。

picture-gallery-style-tests/resources/public/js/site.js

```
function colorStr(color) {
    return "rgb("+color[0]+","+color[1]+","+color[2]+")";
}
function setColor(div, colors) {
    var bgColor = colors[0];
    var textColor = colors[1];
    div.css("background-color", colorStr(bgColor));
    div.find('a').css("color", colorStr(textColor));
}
```

① https://github.com/chengyin/albumcolors

```
$(document).ready(function(){
    $(".thumbnail")
            .each(function() {
            var div = $(this);
            var url = div.find('img').attr('src');
            var thumbColors = new AlbumColors(url);

            var color = "";
            thumbColors.getColors(function(colors) {
                setColor(div, colors);
            });
        });
});
```

当我们重新加载页面就会看到这些变化，每个缩略图的背景应该被设为其主色调，并且其文字颜色被设为对应的补色。相册页面如图 6-1 所示。

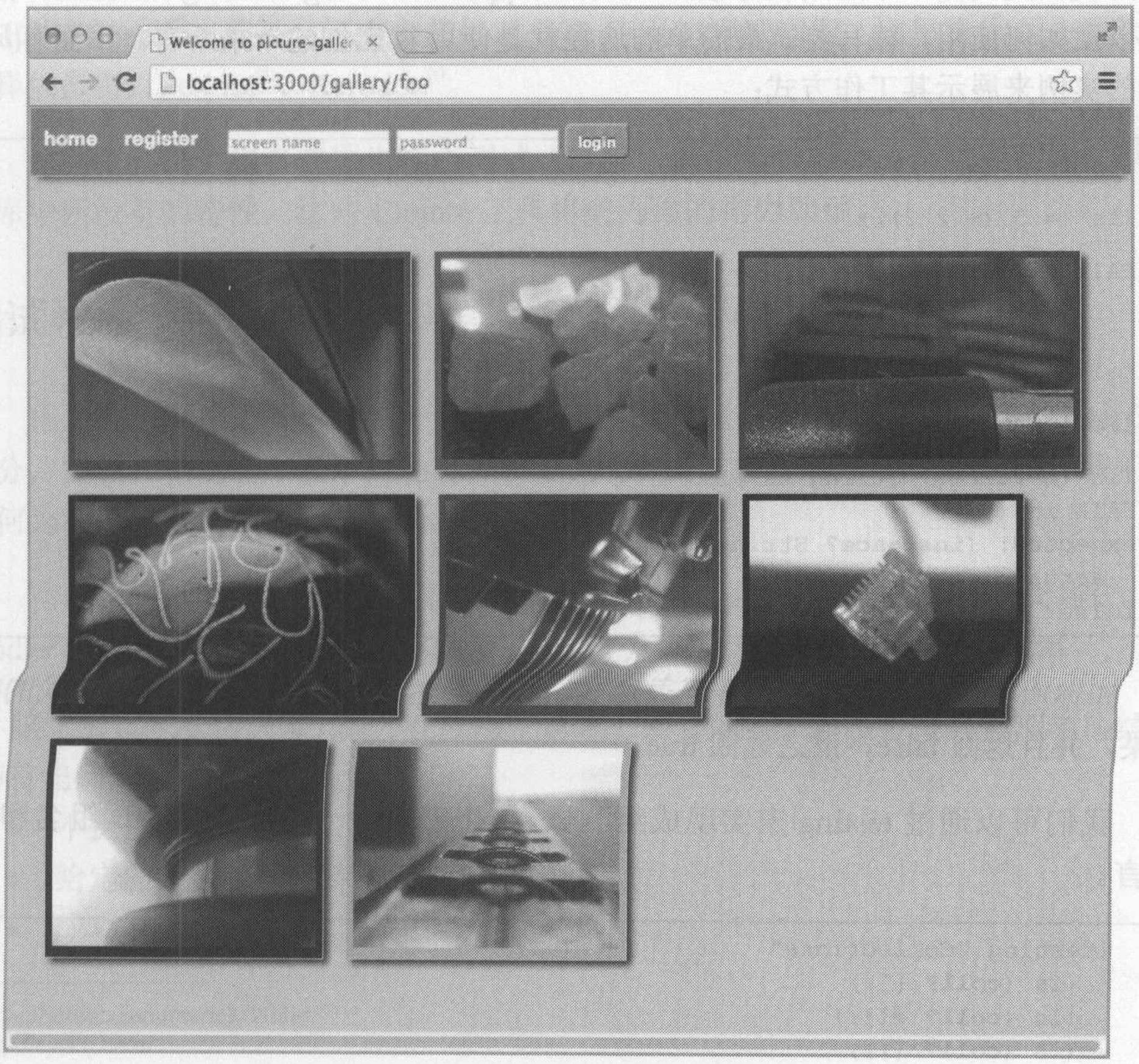

图 6-1 样式化处理的相册

6.2　单元测试

有很多软件方法论研究软件测试的方式、方法、时机。在很多人看来这是个极为敏感的话题。正因如此，我只对测试工具进行一些基本概述，关于要以何种方式并在何时使用，由你自己决定。

测试 API

Clojure 在 clojure.test 命名空间中提供内建测试工具支持。在工程建立之初，测试包同时就被生成了。

那我们先浏览一下应用程序接口（API，Application Programming Interface）的模样，以及如何使用。编写测试最简单的途径就是使用 is 宏创建一条断言。下面有几则简单的实例来展示其工作方式：

```
(is (= 4 (+ 2 2)))

(is (= 5 (+ 2 2)))
FAIL in (:1)
    expected: (= 5 (+ 2 2))
    actual: (not (= 5 4))
false

(is (even? 2))

(is (instance? String 123))
FAIL in (:1)
expected: (instance? String 123)
  actual: java.lang.Long
false
```

如你所见，is 宏可以接受任何表达式。如果表达式错误，则会打印表达式的实际结果，并且返回 false，反之返回 true。

我们可以通过 testing 宏实现成组测试。这个宏接受一个字符串测试组名和多个断言。

```
(testing "Collections"
  (is (coll? {}))
  (is (coll? #{}))
  (is (coll? []))
  (is (coll? '())))
```

最终，我们可以使用 deftest 定义测试：

```
(deftest collections-test
  (testing "Collections"
    (is (coll? {}))
    (is (coll? #{}))
    (is (coll? []))
    (is (coll? '()))))
```

使用 deftest 定义的测试可以被普通函数调用。你一样可以在 REPL 中调用 run-tests 运行测试。程序 test 文件夹下的所有测试可以通过 Leiningen 运行，执行 lein test 即可。虽然 API 还包含几个别的辅助函数，不过，我希望前面几个实例已经让你对其有了认识，足以开始使用。

最后，很值得一提的是，在 Clojure 的生态圈中还有不少测试框架可以用，比如 Midje[①]、Speclj[②]。此外，还有专门针对 Web 应用的测试框架。这里提供两个受欢迎的选择供参考：Peridot[③]、Kerodon[④]。

这些框架提供很多在原生核心测试 API 所没有的特性。如果你需要某些在原生测试库中所没有的特性，这些 Clojure 工具将成为你的得力助手。

测试程序

程序有两种类型的路由：一种是程序用来服务于用户接口（UI，User-interface）部分，完成在浏览器中显示；还有一种是为 UI 操作暴露服务端的 handler。我们看看如何为程序的登录处理环节编写测试。

在应用中已经有了一个测试模板，你可以在 test/picture_gallery/test/中找到它。测试 handler 名为 handler.clj。打开它你就会看到其中定义了一个名为 test-app 的测试。

这个测试现在会失败，因为在应用中请求/URI 的时候，还没有按其预期响应结果，我们首先确定想要测试的场景：

- 不提供参数登录。
- 给定的参数在数据库中无法匹配用户。
- 成功登录。

① https://github.com/marick/Midje
② http://speclj.com/
③ https://github.com/xeqi/peridot
④ https://github.com/xeqi/kerodon

在程序中，请求路由可以使用如下代码：

```
(app (request <method> <url> <params>))
```

这个响应是标准的 Ring 响应，其内容详见第 2 章（“Clojure 的 Web 技术栈”，第 23 页）。

我们想要访问/login URL，并发送用户 ID 和密码。可是，我们仅希望测试请求 handler 而不是模型。测试应该依赖当前存放在数据库中的账户。

很可惜，picture-gallery.routes.auth 命名空间中的 handle-login 函数访问 picture-gallery.models.db 命名空间中的 get-user。当我们在测试中调用此函数，实际是查询数据库中的用户。

在一些语言中，可以使用“猴子补丁”（Monkey Patching）来解决这个问题。这种方法允许你在运行时使用自己的版本简单重写不合适的函数。其弊端是这种方式的修改是全局性的，有可能与原始版本的代码难以协作。

Clojure 提供的 with-redefs 宏在其代码作用域范围内重定义 Vars。这种手段让我们可以通过帅气、可靠的方式在运行时修改，让我们真正知道什么是代码影响。

而我们的目的正是在测试范围中通过 mock 函数重定义 get-users 函数。这便于我们不必事先在编写程序业务逻辑时考虑测试的问题。那么我们通过实例来看看如何使用。我们首先定义一个 mock 函数，此函数返回一个测试用户。

picture-gallery-style-tests/test/picture_gallery/test/handler.clj

```
(defn mock-get-user [id]
  (if (= id "foo")
    {:id "foo" :pass (encrypt "12345")}))
```

这里，我们同样需要引用 noir.util.crypt/encryp 来对密码明文进行加密操作。

picture-gallery-style-tests/test/picture_gallery/test/handler.clj

```
(ns picture-gallery.test.handler
  (:require [clojure.test :refer :all]
            [ring.mock.request :refer :all]
            [noir.util.crypt :refer [encrypt]]
            [picture-gallery.handler :refer :all]))
```

我们现在可以将 picture-gallery.models.db/get-user 重定义为 mock 函数，然后再运行测试：

```
(with-redefs [picture-gallery.models.db/get-user mock-get-user]
  (app (request :post "/login" {:id "foo" :pass "12345"})))
```

当我们在 REPL 中运行以上代码，我们会看到返回一个重定向及携带会话 ID 的 cookie：

```
{:status 302
 :headers {"Set-Cookie" ("ring-session=0645d310-892b-43c0-a4d5-dcaa87859a67;Path=/")
           "Location" "/"}
 :body ""}
```

现在，我们可以运行一个用例来测试找不到账户的情况：

```
(with-redefs [picture-gallery.models.db/get-user mock-get-user]
  (app (request :post "/login" {:id "bar" :pass "12345"})))
```

这次就没有生成会话，并且被简单重定向到程序的“/” URL：

```
{:status 302
 :headers {"Set-Cookie" ()
           "Location" "/"}
 :body ""}
```

那接下来，我们将以上代码汇总，编写一个用在程序登录环节的单元测试。

picture-gallery-style-tests/test/picture_gallery/test/handler.clj

```
(ns picture-gallery.test.handler
  (:require [clojure.test :refer :all]
            [ring.mock.request :refer :all]
            [noir.util.crypt :refer [encrypt]]
            [picture-gallery.handler :refer :all]))
(defn mock-get-user [id]
  (if (= id "foo")
    {:id "foo" :pass (encrypt "12345")}))
(deftest test-login
  (testing "login success"
    (with-redefs [picture-gallery.models.db/get-user mock-get-user]
      (is
        (-> (request :post "/login" {:id "foo" :pass "12345"})
          app :headers (get "Set-Cookie") not-empty))))
  (testing "password mismatch"
    (with-redefs [picture-gallery.models.db/get-user mock-get-user]
      (is
        (-> (request :post "/login" {:id "foo" :pass "123456"})
          app :headers (get "Set-Cookie") empty?))))
  (testing "user not found"
    (with-redefs [picture-gallery.models.db/get-user mock-get-user]
      (is
        (-> (request :post "/login" {:id "bar" :pass "12345"})
          app :headers (get "Set-Cookie") empty?)))))
```

6.3　日志

现实中，几乎所有的应用都需要某种日志功能。对于程序，至少需要对产生的错误进行记录。这有助于我们在生产环节对错误进行跟踪和重现。

有不少日志工具都可以使用。在本章中，我们使用 Timbre 库[①]。首先，我们必须在项目描述文件中加载 Timbre 库。

```
picture-gallery-logging/project.clj
[com.taoensso/timbre "2.6.1"]
```

这样，我们在需要的命名空间中引用就可以试用了，其函数有 trace（跟踪）、info（信息）、warn（警告）、debug（调试）和 fatal（错误）。我们这就用在 upload 命名空间，当我们尝试删除文件而抛出异常时，就记录一条错误日志。

```
(nspicture-gallery.routes.upload(:require...
    [taoensso.timbre
        :refer[trace debug info warn error fatal]]

picture-gallery-logging/src/picture_gallery/routes/upload.clj
(defn delete-image [userid name]
  (try
    (db/delete-image userid name)
    (io/delete-file (str (gallery-path) File/separator name))
    (io/delete-file (str (gallery-path) File/separator thumb-prefix name))
    "ok"
    (catch Exception ex
      (error ex "an error has occured while deleting" name)
      (.getMessage ex))))
```

这样就行了。为了验证这一点，我们可以上传一张图片，然后手工在文件夹中将其删除。最后，我们在界面上进行删除操作，就会触发异常。这个异常会被日志记录。

Timbre 的配置由包含特定键名的 vector 来表达，其代表具体配置项的路径。举个例子，如果我们想为日志配置自定义的时间戳格式，可以通过调用 set-config!进行配置。

我们打开 picture-gallery.handler 命名空间并添加 Timbre 的引用。

```
(ns picture-gallery.handler
  (:require ... [taoensso.timbre :as timbre]))
```

① https://github.com/ptaoussanis/timbre

对现在的 init 函数，我们做如下初始化配置：

```
(timbre/set-config! [:timestamp-pattern] "yyyy-MM-dd HH:mm:ss")
```

我们还可以使用配置重定向日志输出到指定的文件。默认情况下，所有的日志会以标准输出结束。可是，通过对日志进行指定输出，我们就可以导出特定类型的日志。

对于指定一个输出，我们需要在配置中添加到 appenders 路径，比如下面的例子：

```
(timbre/set-config!
  [:appenders :info-appender]
  {:min-level :info
  :enabled? true
  :async? false
  :max-message-per-msecs 100
  :fn info-appender})
```

error-appender 是个接受 map 的函数，其包含如下键名：:ap-config、:level、:prefix、:message 和:more。一个简单的输出函数可能看起来如下：

```
(defn info-appender [{:keys [level message]}]
  (println "level:" level "message:" message))
```

我们现在就可以将 println 替换成适当的日志函数：

```
(defn init []
  (timbre/set-config!
   [:appenders :info-appender]
   {:min-level :info
    :enabled? true
    :async? false
    :max-message-per-msecs 100
    :fn info-appender
    })
  (timbre/info "picture-gallery started successfully"))

(defn destroy []
  (timbre/info "picture-gallery is shutting down"))
```

当服务启动之后，我们将会看到类似以下内容的日志：

```
2013-May-04 12:31:12 -0400 Helios.local INFO [picture-gallery.handler]
 - picture-gallery started successfully
```

现实中更可能使用现成的输出，比如 rotor。rotor 是个回环日志文件输出，通过添加如下依赖项实现。

picture-gallery-logging/project.clj

```
[com.postspectacular/rotor "0.1.0"]
```

我们现在可以为 rotor 创建配置，这有点类似于我们前面做的工作。

```
(ns picture-gallery.handler
  (:require ...
    [taoensso.timbre :as timbre]
    [com.postspectacular.rotor :as rotor]))
```

picture-gallery-logging/src/picture_gallery/handler.clj

```
(defn init []
  (timbre/set-config!
    [:appenders :rotor]
    {:min-level :info
     :enabled? true
     :async? false ; should be always false for rotor
     :max-message-per-msecs nil
     :fn rotor/append})
  (timbre/set-config!
    [:shared-appender-config :rotor]
    {:path "error.log" :max-size (* 512 1024) :backlog 10})

  (timbre/info "picture-gallery started successfully"))
```

注意，rotor 输出需要独立配置。我们提供一个路径，一个回写最大值，以及一个保留旧文件的数量：

```
(set-config!
  [:shared-appender-config :rotor]
  {:path "/var/log/error.log" :max-size (* 512 1024) :backlog 10})
```

我们现在有个适合我们程序的日志配置。在生产阶段产生的任何错误，我们可以回滚并查看触发原因。这有助于我们发现原因并修复它们。

现在，我们已经完成了为之兴奋的相册程序，再来看看如何发布。由于程序运行在 Java 虚拟机上，我们有很棒的部署选择，甚至可以将其交给主机供应商、一个虚拟专用服务器或者一个云服务。

6.4 程序配置文件

在很多情况下，你可能需要使用不同的配置运行你的程序，例如，你的程序可能依赖于设置的变量，比如运行的端口以及数据库连接信息。当你在开发阶段本地运行，这些变量很可能不同于在生产环节中的机器。

解决这个问题的一个常见的做法就是通过在程序外设置环境变量，在运行时读取它来实现。在本节，我们将看到如何从 Leiningen 配置文件或是系统环境中读取这些变量。

Leiningen 配置文件允许管理应用程序的方方面面，比如依赖项、资源路径、环境变量。你可以为不同的场景创建不同的配置文件，并且当程序通过配置文件运行，它将得到当前有效的环境信息。

配置文件可以直接写在 project.clj 当中，也可以放在同目录下一个独立的 profiles.clj 文件中。假如配置中存在某种敏感信息，而不想将其提交到版本控制的时候，后者会更实用。并且 profiles.clj 配置优先级高，可以覆盖 project.clj。

接下来，我们打开相册的 project.clj 配置文件。你应该注意到，它已经对开发和部署分别做了配置：

picture-gallery-logging/project.clj

```
:profiles
{:production
 {:ring
  {:open-browser? false, :stacktraces? false, :auto-reload? false}}
 :dev
 {:dependencies [[ring-mock "0.1.5"] [ring/ring-devel "1.2.0"]]}}
```

我们更新这些配置项，添加环境变量：

```
:profiles
{:production
  {:ring {:open-browser? false,
          :stacktraces? false,
          :auto-reload? false}
   :env {:port 3000
         :db-url "//localhost/gallery"
         :db-user "admin"
         :db-pass "admin"
```

```
            :galleries-path "galleries"}}
    :dev
    {:dependencies [[ring-mock "0.1.5"]
                    [ring/ring-devel "1.2.0"]]
     :env {:port 3000
           :db-url "//localhost/gallery"
           :db-user "admin"
           :db-pass "secretProdPasword"
           :galleries-path "galleries"}}}
```

我们使用 Environ 从配置文件去读取配置变量。要使用它，我们首先要在项目中添加如下依赖项和插件：

```
:dependencies [... [environ "0.4.0"]]
:plugins [... [lein-environ "0.4.0"]]
```

现在我们可以更新 picture-gallery.models.db 命名空间，从环境变量读取数据库配置：

```
(ns picture-gallery.models.db
  (:require ...
            [environ.core :refer [env]]))

(def db
  {:subprotocol "postgresql"
   :subname (env :db-url)
   :user (env :db-user)
   :password (env :db-pass)})
```

我们还可以更新 picture-gallery.util/galleries 从环境变量读取路径变量。

```
(ns picture-gallery.util
  (:require [noir.session :as session]
            [hiccup.util :refer [url-encode]]
            [environ.core :refer [env]])
  (:import java.io.File))

...

(def galleries (env :galleries-path))
```

不幸的是，如果你使用的是 Eclipse，Counterclockwise 插件不会从配置文件加载变量。还好，Environ 可以直接读取环境变量。

对于添加变量，最简单的是在 Eclipse 中打开运行配置，选择项目。我们可以在环境变量列表中添加变量。下一次我们启动 REPL 时，环境变量会生效，Environ 就能读取了。

6.5 打包应用

程序现在可以开始打包部署了。在本节，我们讨论如何使用 Leiningen 完成。如你所见，这是个直截了当的过程，尽管如此，你还有几条注意事项，这取决于你将选择什么发布工具来运行。

到目前为止，我们都是在 REPL 中运行程序，或者在开发模式中通过调用 lein ring 服务去启动 Jetty。在这种模式下，服务会观察文件变更并根据需要重新加载。这显然会严重影响性能。

在生产环节，这里提供两种方式运行 Clojure 的 Web 应用。我们来看看两者的利弊。

第一种方式是创建一个可独立运行的执行程序，比如 Jetty 的嵌入服务。这种途径的应用不会需要任何额外的依赖项，可以直接在系统中运行。

这种方式的缺点是我们不得不手动配置所有的服务配置以及日志、数据库连接、SSL 配置等。这还意味着每个应用程序会产生更多的开销，我们需要为此部署独立的服务器。

第二种方式是创建一个 Web 应用程序存档（WAR，Web Application Archive），那么就能部署在类似 Tomcat 这种应用服务中。在这种方式中，我们可以将与环境配置相关的所有内容放在应用服务中。当程序部署完毕，它们就能通过环境变量来读取此配置。

应用服务还支持通过共享域名部署多个应用。这允许我们为每个应用付出更小的开销，还能为部署在此服务的这些应用提供通用配置。这种容器可以持续追踪数据库连接设置、日志配置、管理 HTTPS 监听等。

这种途径对于开发、持续交付、上线阶段有着不同配置或者管理多个应用尤为便捷。既然程序内部没有配置，你就不需要为部署在不同的阶段去隔离配置。

这种途径的缺点是应用程序服务的发布会比嵌入 Jetty 要多一些开销。要知道，程序服务的配置通常会更复杂。这种工作显得不是太合理，这取决于你计划在生产阶段如何管理程序。

好处是这两种部署方式都容易打包程序，如果你使用其中之一，要想切换成另外一种也不费太大劲。

独立部署

接下来，我们看看部署独立程序所涉及的更详尽的内容。

运行 uberjar

当你希望将程序打包成可独立运行的发布版本，我们可以在项目根目录简单运行以下命令：

```
lein ring uberjar
```

目标文件会生成到 target 文件夹下，我们现在可以输入以下命令运行此 Jar 包——java -jar picture-gallery-0.1.0-SNAPSHOT-standalone.jar。一旦服务启动之后，你就可以通过浏览器打开 localhost:3000 来浏览。

此服务默认运行在 3000 端口，也可以通过设置$PORT 环境变量覆盖默认端口设置。

通过 HTTP Kit 运行

uberjar 会通过嵌入 Jetty 服务来创建。尽管如此，还是有托管容器可以接受 Jetty，比如 HTTP Kit。

HTTP Kit 是个在 Clojure 的 Ring 兼容的事件驱动服务，旨在用来取代 Jetty。不同于 Jetty，HTTP Kit 使用非阻塞 I/O 模型处理请求。这将提供极高的吞吐量和极强的伸缩能力。

在使用 HTTP Kit 前，我们需要在 project.clj 中引用依赖项，并创建 main 方法。

```
:dependencies [... [http-kit "2.1.12"]]
```

lein ring 现在还不支持 HTTP Kit，我们还需要创建 main 函数才能运行。我们来新建一个名为 picture-gallery.main 的命名空间并添加如下内容：

```
(ns picture-gallery.main
  (:use picture-gallery.handler
        [org.httpkit.server :only [run-server]]
        [ring.middleware file-info file])
```

```
  (:gen-class))
(defn -main [& [port]]
  (let [port (if port (Integer/parseInt port) 3000)]
    (run-server app {:port port})
    (println (str "You can view the site at http://localhost:" port))))
```

这里-main 函数简单实现了一个 HTTP Kit 服务接口，并通过调用 org.httpkit.server/run-server 传入 app handler。这里唯一需要注意的是，我们必须在此命名空间的声明处使用:gen-class 标记，这样才能正确编译成 Java 字节码。

注意此处-main 函数的“-”前缀说明此函数供 Java 访问。这么做是必须的，用于 Java 独立程序的入口函数。

下一步，我们在 project.clj 中简单描述 main 方法，添加如下指令：

```
:main picture-gallery.main
```

现在可以编译生成可独立运行的程序了，再运行以下代码：

```
lein uberjar
java -jar target/picture-gallery-0.1.0-SNAPSHOT-standalone.jar
```

整个过程就是这样。如果你的应用需要额外的功能或任何 HTTP Kit 特性，你都可以简单替换掉 Jetty。

通过 leiningen trampoline 运行

还有一种运行程序的方式，就是在控制台通过 lein 运行程序之后，使用 lein trampoline 可以将其终止。我们可以运行如下代码：

```
lein trampoline ring server-headless PORT
```

这里的 PORT 必须作为一个环境变量提供。

这种方法要使用 project.clj 中:ring 键下的配置说明。举个例子，相册有如下配置：

```
:ring {:handler picture-gallery.handler/app
       :init picture-gallery.handler/init
       :destroy picture-gallery.handler/destroy}
```

lein trampoline 对使用 Leiningen 控制程序的完整生命周期进行了补充。

通过 daemon 运行

作为*nix 系统中的 daemon，我们可以简单运行程序。例如，在 Ubuntu 上，我们可以创建一个 upstart[①]配置。我们只用创建一个/etc/init/gallery.conf 配置文件，并添加如下设置即可：

```
## Upstart config file (use 'start gallery', 'stop gallery')
## stdout and stderr will be captured in /var/log/upstart/gallery.log
author "Me"
description "Start the Picture Gallery webapp on its default port"
start on (local-filesystems and net-device-up IFACE!=lo)
exec java -jar /srv/gallery/picture-gallery-0.1.0-SNAPSHOT-standalone.jar

## Try to restart up to 10 times within 5 min:
respawn limit 10 300
```

应用服务部署

我们现在知道怎么独立运行应用程序了，再对比看看如何运行在应用服务上。

Tomcat 部署

要将你的应用部署在 Tomcat 上，首先需要下载一份 Tomcat 服务的拷贝，并将其文件提取到本地。

通过运行 Tomcat 目录下的 bin/catalina.sh，你就能启动 Tomcat。你可以看见位于 logs/catalina.out 的服务日志。当服务启动之后，你应该在日志中看见如下类似内容：

```
May 5, 2013 11:12:25 AM org.apache.catalina.core.AprLifecycleListener init
INFO: The APR based Apache Tomcat Native library which allows optimal performance
in production environments was not found on the java.library.path:
.:/Library/Java/Extensions:/System/Library/Java/Extensions:/usr/lib/java
May 5, 2013 11:12:25 AM org.apache.coyote.AbstractProtocol init
INFO: Initializing ProtocolHandler ["http-bio-3000"]
...
```

① http://upstart.ubuntu.com/

停止服务与启动操作类似，运行 bin/catalina.sh 即可。要将应用以 WAR 包发布，我们只需要使用 uberwar 参数即可：

```
lein ring uberwar
```

此命令可手工生成可部署的 WAR 包，并能将其部署在应用服务中。部署应用首先要将生成的文件拷贝到 Tomcat 的 webapps 文件夹中。

对于部署的应用程序，上下文存在一个相对服务根目录的描述。通常此上下文大都是文件名。我们将部署文件命名为 picture-gallery.war。

```
cp target/picture-gallery-0.1.0-SNAPSHOT-standalone.war \
~/tomcat/webapps/picture-gallery.war
```

应用程序现在应该就能用了，可以打开 http://localhost:3000/picture-gallery 看看。

注意，还是有可能需要在服务的根目录部署引用。如果这样的话，就将文件命名为 ROOT.war。

Immutant 部署

部署还有个极好的选择，那就是 Immutant。这是个专门为 Clojure 设计的托管应用服务，并且还提供自主集成的 Leiningen 工具。我们可以通过 Leiningen 安装 Immutant，运行如下命令：

```
lein immutant install 1.0.2
```

一旦安装了 Immutant，我们可以在程序根目录简单运行如下命令：

```
lein immutant deploy
```

启动 Immutant 服务同样容易，使用如下命令：

```
lein immutant run
```

我们讨论过的这些方式都需要我们自己维护服务。还有一种办法就是将应用部署到云服务上。Heroku 是明确说明了支持运行 Clojure 应用程序的云服务。在下一节，我们会看看如何将应用部署在云服务上去。

Heroku 部署

Heroku 云服务提供了免费可选套餐。在使用 Heroku 之前，我们首先要确认已经安装了 Git[①]和 Heroku Toolbelt[②]。

Heroku 使用的命令行启动应用程序，具体命令由名为 Procfile 的文件指定。这个文件必须放在项目的根目录。

在 Heroku 上运行 Clojure 程序通常都是通过使用 Leiningen 加 trampoline 参数，这个我们之前讨论过。首先我们要在 Procfile 中添加如下命令：

```
web: lein with-profile production trampoline ring server
```

下一步我们需要初始化这个程序的 Git 仓库，通过运行如下命令：

```
git init
git add .
git commit -m "init"
```

当仓库建立之后，我们可以通过运行 foreman start 测试应用。如果程序运行正常，我们就可以考虑通过如下命令将程序部署到云服务上：

```
heroku create
```

我们运行这条命令，为程序提供 Postgres 支持：

```
heroku addons:add heroku-postgresql
```

在 Heroku 控制面板，你可以找到关于数据库的连接设置。你需要为应用程序添加数据库配置。我们现已准备好了将应用上传到 Heroku。

```
git push heroku master
```

当上传成功之后，Heroku 会试图建立并部署你的应用。如果此环节顺利完成，通过管理员控制台中指定的 URL，就能使用浏览器访问应用了。

① http://git-scm.com/
② https://toolbelt.heroku.com/

6.6 你学到什么

这便是如何设计并实现，最终部署站点。在构建和设计过程中，我们从现实环境出发，考虑到了很多方面，例如处理静态资源、访问数据库及使用 Ajax。

但愿你对这些步骤能融汇吸纳，并能通过 Clojure 在现实中建立实际的应用。

我们网站的功能明显还能做进一步的改进。你可能希望对于大型相册考虑实现分页，为每个用户提供多套相册，能一次性上传多张图片，为上传的图片设置可见权限。

到目前为止，我们仅涉及一组用于开发 Web 程序的库。尽管如此，Clojure 的 Web 栈是非常灵活的，并且对库的替换代价也不大。在下一章，我们会涉及如何通过 Selmer 模板引擎、ClojureScript、SQL Korma 重构相册。

第7章

混合

在本章中，我们会见识到通过几个其他方式去实现第5章的“相册”程序（第80页）。我们也会见识到如何使用ClojureScript替换HTML模板引擎，以及使用Korma领域特定语言（DSL，Domain-specificlanguage）编写SQL查询语句。

7.1 使用Selmer

在第2章(Clojure的Web技术栈,第23页),我提到有不少模板语言可以用在Clojure。在本节，我们来看看Selmer[①]引擎，并了解如何使用Hiccup实现移植相册的功能。

之所以选择Selmer而不是别的类似流行库，一方面是因为虽然它跟Enlive很像，但相对熟悉且容易上手。Enlive的学习曲线比较陡，掌握起来会比较难。

另一方面，Selmer是基于Django[②]模板引擎。如果你熟悉Django或者类似的模板语言，你应该会觉得很亲切。使用Selmer的另外一个优势就是速度快。

Selmer有什么区别？

Hiccup是个优美且简单的模板引擎。不幸的是，最强的优势也是最大的不足，由于Hiccup使用的是Clojure数据结构，它们仅仅是普通代码的一部分。

① https://github.com/yogthos/Selmer
② https://docs.djangoproject.com/en/dev/ref/templates/

这使得对 Clojure 不熟悉的人无法真正模板化，毕竟不是给个想法就能让模板为你干活的。

还有个问题，如果你不格外注意，它会使得前端到后端的逻辑相互渗透。最终，你每次修改一个布局，都不得不重新部署网站。

Selmer 提供一个强力的微语言（Mini Language），能用来实现通用模板。当用来生成 HTML 模板，鼓励从描述上和业务逻辑上彻底分离。除此之外，模板化的好处就是可以让即使完全不懂 Clojure 的人也能维护。

创建模板

模板是简单的 HTML 文件，外加一些模板标签。我们来看看一份模板的例子：

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
    <title>My First Template</title>
  </head>
  <body>
    <h2>Hello {{name}}</h2>
  </body>
</html>
```

通过 map 表示上下文，用来显示模板内容。在模板中可以使用上下文包含的任何需要的变量。在前面的代码中，我们有个模板化的页面，并提取和使用了名为 name 的变量。

这里有两个函数用来显示模板，名为 render 和 render-file。render 函数接受一个代表模板的字符串。render-file 接受一个表示模板文件路径的字符串。

如果我们之前保存了定义的模板名为 index.html，那么我们可以如下显示：

```
(ns example.routes.home
  (:require [selmer.parser :refer [render-file]]))

(defn index [request]
  (render-file "example/views/templates/index.html"
               {:name "John"}))
```

render-file 函数的第一个参数用来表示模板文件的路径。此路径应该是程序的 src 目

录下的相对路径。第二个参数是map代表了模板的上下文。

在此前的代码中，我们通过变量名获取字符串值。不仅如此，我们还不能限制仅为字符串，而是我们想要的任何格式，比如我们可以使用for标签通过如下方式使用集合类型：

```
<ul>
{% for item in items %}
<li> {{item}} </li>
{% endfor %}
</ul>

(render-file "/example/views/templates/items.html
              {:items (range 10)})
```

如果碰到的子项是map类型，我们还可以按键名处理：

```
(render "<p>Hello {{user.first}} {{user.last}}</p>"
        {:user {:first "John" :last "Doe"}})
```

如果在模板中没有指定特殊的操作，那么会调用参数的.toString获取结果。

使用 filter

filter允许在显示前对变量进行预处理。比如，你可以使用filter将变量转换为大写、计算哈希或是计算长度。filter是在变量名的后面使用管道符号“|”表达：

```
{{name|upper}}
```

Selmer提供了很多方便的filter，比如upper（大写）、date（日期）、pluralize（名词复数），开箱即用。此外，我们还可以很容易使用selmer.filters或add-filter!定义我们自己的filter。

```
(add-filter! :empty? empty?)

(render "{% if files|empty? %}no files{% else %}files{% endif %}"
  {:files []})
```

通常情况下，filter的上下文会遗漏，我们可以使用以下代码覆盖上面的行为：

```
(add-filter! :foo
  (fn [x] [:safe (.toUpperCase x)]))

(render "{{x|foo}}" {:x "<div>I'm safe</div>"})
```

使用模板标签

Selmer 提供两种类型的标签。第一种是内联标签，类似 extends 和 include。这些标签是独立（self-contained）声明，并且不需要结束标签。另外一种是块标签，这种类型的标签有起止标示，功能操作都在文本块中。最具代表的例子应该是 if...endif。

自定义标签

除了已经提供的这些标签，你还可以通过使用 selmer.parser/add-tag!宏轻松自定义标签，我们来看看使其工作的例子：

```
(add-tag!
:image
(fn [args context-map]
   (str "<img src=" (first args) "/>")))
(render "{% image \"http://foo.com/logo.jpg\" %}" {})
```

我们还可以通过 add-tag!重载块标签。这时，我们需要提供操作标签，后面跟随处理函数和结束标签。处理函数接受一个 addition 参数，并持有块中的上下文。如下面的例子，content 由块的键名组成：

```
(add-tag! :uppercase
          (fn [args context-map content]
            (.toUpperCase (get-in content [:uppercase :content])))
          :enduppercase)

(render "{% uppercase %}foo {{bar}} baz{% enduppercase %}" {:bar "injected"})
```

继承模板

Selmer 模板可以通过使用 block 标签来引用别的模板。调用模板有两种途径：可以通过使用 extends 标签来扩展模板，或是通过使用 include 标签来加载模板。

扩展模板

当我们使用 extends 标签的时候，当前模板会以被引用的模板为基础。而被引用的模板名字与当前模板一致则会被覆盖。

那我们来看一则具体的例子。首先，我们定义基础模板并将其命名为 base.html：

```
<!DOCTYPE html>
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{% endblock %}</title>
</head>
<body>
    <div id="content">
        {% block content %}{% endblock %}
     </div>
</body>
</html>
```

接下来，我们创建一个新的模板，名为 home.html，通过如下方式对 base.html 扩展：

```
{% extends "base.html" %}

{% block content %}
    {% for entry in entries %}
        <h2>{{ entry.title }}</h2>
        <p>{{ entry.body }}</p>
    {% endfor %}
{% endblock %}
```

当显示 home.html 的时候，content 块会显示定义的 entries，但是我们并没有在这里定义 title，这就会使用 base.html 中定义的那个。

还有，你可以将模板串在一起。这样最后一个例子中的块标签就可以用一个块来处理。

包含模板

include 标签允许为当前模板加载另外一个模板的内容。我们来看一个例子。假设我们有个 base.html 模板，其中包含名为 register.html 和 home.html 的两个模板，下面

定义名为 register 和 home 的块：

```
<!DOCTYPE html>
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block title %}My amazing site{% endblock %}</title>
</head>
<body>
    <div id="content">
        {% if user %}
            {% include "home.html" %}
        {% else %}
            {% include "register.html" %}
        {% endif %}
    </div>
</body>
</html>
```

我们现在可以为 home.html 和 register.html 两个模板文件分别定义内容了：

```
<h1>Hello {{user}}</h1>
```

```
<form action="/register" method="POST">
    <label for="id">user id</label>
    <input id="id" name="id" type="text"></input>
    <input pass="pass" name="pass" type="text"></input>
    <input type="submit" value="register">
</form>
```

当 base.html 在加载使用的时候，会将 include 标签内容替换成被包含模板文件的内容。

将相册转换成 Selmer

我们已经见识到了，Selmer 使用纯 HTML 去定义页面。我们首先要确认应用中负责显示页面的部分并将其做对应转换。

那我们来看看应用中已有的路由，确切说是那些负责为页面提供服务的路由。

home 命名空间只包含一条显示主页的路由：

- GET "/"

gallery 命名空间有一条为具体用户显示相册的路由：

- GET "/gallery/:userid"

upload 命名空间包含的路由是用来上传和删除照片的。只有一条路由负责服务于显示页面：

- GET "/upload"
- POST "/upload"
- POST "/delete"

最后还有 auth 命名空间，它包含的路由是用来注册、身份验证以及账户管理的：

- GET "/register"
- POST "/register"
- POST "/login"
- GET "/logout"
- GET "/delete-account"
- POST "/confirm-delete"

如你所见，只有少量路由是负责显示页面的。这使得我们的任务相对简单了。

我们现在打开 project.clj，并往其中添加 Selmer“[selmer "0.5.4"]”依赖项。

下一步，我们创建一个名为 picture-gallery.views.templates 的命名空间，这用来存放 Selmer 的所有模板。

我们当前的布局使用的是 Hiccup 创建的骨架，并在页面中使用。现在我们可以更新 layout 命名空间使用它们来显示模板。

当你再次调用时，Selmer 使用 HTML 模板和 map 参数处理。我们需要将替换布局中的通用辅助函数改为 render 函数，这样就可以处理显示模板。做这些之前我们还需要在命名空间定义处添加对 selmer.parser 命名空间的引用。

```
(ns picture-gallery.views.layout
    (:require [selmer.parser :as parser]))
```

现在，我们定义了模板的路径并编写了辅助函数用于显示模板。这个函数用来负责对模板路径预处理、添加 Servlet 负载键、设置会话用户。

在布局中放置好样板文件之后，我们就可以专注于与任务相关的参数，到解析模板时就能用上了。

picture-gallery-selmer/src/picture_gallery/views/layout.clj

```
(def template-folder "picture_gallery/views/templates/")

(defn utf-8-response [html]
  (content-type (response html) "text/html; charset=utf-8"))

(deftype RenderablePage [template params]
 Renderable
 (render [this request]
  (->> (assoc params
              :context (:context request)
              :user    (session/get :user))

       (parser/render-file (str template-folder template))
       utf-8-response)))

(defn render [template & [params]]
  (RenderablePage. template params).)
```

现在，我们来更新布局，为页面创建基础模板。我们将其命名为 base.html 并放置在 src/picture_gallery/views/templates/目录中。

picture-gallery-selmer/src/picture_gallery/views/templates/base.html

```
<html>
<head>
<title>Welcome to picture-gallery</title>
<link href="{{context}}/css/screen.css" rel="stylesheet" type="text/css" />
<script type="text/javascript">
    var context = "{{context}}";
</script>
<script src="//code.jquery.com/jquery-2.0.2.min.js" type="text/javascript"></script>
<script src="{{context}}/js/colors.js" type="text/javascript"></script>
<script src="{{context}}/js/site.js" type="text/javascript"></script>
</head>
<body>
  {% block menu %}
  <div id="usermenu">
    <div class="menuitem"><a href="/">home</a></div>
    {% if user %}
    <div class="menuitem">
     <a href="/upload">upload images</a>
    </div>
    <div class="menuitem">
     <a href="/logout">logout {{user}}</a>
    </div>
    <div class="menuitem">
      <a href="/delete-account">delete account</a>
     </div>
    {% else %}
    <div class="menuitem">
      <a href="{{context}}/register">register</a>
```

```
    </div>
    <div class="menuitem">
      <form action="{{context}}/login" method="POST">
        <input id="id"
               name="id"
               placeholder="screen name"
               type="text">
        <input id="pass"
               name="pass"
               placeholder="password"
               type="password">
          <input type="submit" value="login">
        </form>
      </div>
      {% endif %}
    </div>
      {% endblock %}
      {% block content %}
      {% endblock %}
</body>
</html>
```

在 Hiccup 版本的程序中，这个基础模板充当了通用函数。它创建基础布局并包含必要的资源。页面现在可以扩展模板，并且添加内容到内容块。注意，在程序部署在应用服务中，我们需要使用 Servlet 上下文，确保可以访问本地资源。

转换主页

我们来新建一个名为 home.html 的模板，更新主页来使用这个模板去显示内容。home 模板会扩展我们之前创建的 base，那么就能创建每个相册的缩略图。这个模板替换 gallery 命名空间下的 show-galleries 函数所实现的功能。

picture-gallery-selmer/src/picture_gallery/views/templates/home.html

```
{% extends "picture_gallery/views/templates/base.html" %}
{% block content %}
<div class="gallery">
  {% for gallery in galleries %}
  <div class="thumbnail">
    <a href="{{context}}/gallery/{{gallery.userid}}">
      <img
        src="{{context}}/img/{{gallery.userid}}/{{thumb-prefix}}{{gallery.name}}"/>
        {{gallery.userid}}'s gallery
    </a>
  </div>
  {% endfor %}
</div>
{% endblock %}
```

随着模板就位，我们现在可以更新 home 路由去使用我们在 layout 中定义的 render 函数。以前，我们使用 gallery 命名空间的 show-galleries 来显示相册。现在我们直接通过在 db 命名空间中获取相册清单并由模板处理。

picture-gallery-selmer/src/picture_gallery/routes/home.clj

```
(ns picture-gallery.routes.home
  (:require [compojure.core :refer [defroutes GET]]
            [picture-gallery.views.layout :as layout]
            [picture-gallery.util :refer [thumb-prefix]]
            [picture-gallery.models.db :as db]
            [noir.session :as session]))
(defn home []
  (layout/render "home.html"
                 {:thumb-prefix thumb-prefix
                  :galleries    (db/get-gallery-previews)}))
(defroutes home-routes
  (GET "/" [] (home)))
```

如果我们重新加载主页，应该可以看见如同之前见到的相册缩略图。

下一步就是创建模板来显示具体相册中的缩略图。首先，我们要在 templates 包中新建一个名为 gallery.html 的文件。这个模板可以在用户相册中显示带对应链接的缩略图。

picture-gallery-selmer/src/picture_gallery/views/templates/gallery.html

```
{% extends "picture_gallery/views/templates/base.html" %}

{% block content %}
<div class="gallery">
   <div id="error"></div>
   <script src="{{context}}/js/gallery.js" type="text/javascript"></script>
   <div>
   {% for pic in pictures %}
      <div class="thumbnail">
         <a class={{pic.name}} href="{{context}}/img/{{pic.userid}}/{{pic.name}}">
         <img src="{{context}}/img/{{pic.userid}}/{{thumb-prefix}}{{pic.name}}"/>
         {% ifequal user page-owner %}
         <input id="{{pic.name}}"
                name="{{pic.name}}"
                type="checkbox"
                value="true" />
         {% endifequal %}
         </a>
</div>
```

```
    {% endfor %}
    {% ifequal user page-owner %}
      <input id="delete" type="submit" value="delete images" />
    {% endifequal %}
    </div>
  </div>
  {% endblock %}
```

如你所见，这个模板非常类似于 home，除了现在这个模板引用了用来管理相册的脚本文件，并且其中缩略图附带的链接是全尺寸图片。

我们现在可以更新 gallery 命名空间，使用 Selmer 模板去显示用户相册。

picture-gallery-selmer/src/picture_gallery/routes/gallery.clj

```
(ns picture-gallery.routes.gallery
  (:require [compojure.core :refer [defroutes GET]]
            [picture-gallery.views.layout :as layout]
            [picture-gallery.util :refer [thumb-prefix]]
            [picture-gallery.models.db :as db]
            [noir.session :as session]))
(defn display-gallery [userid]
  (layout/render "gallery.html"
                 {:thumb-prefix thumb-prefix
                  :page-owner   userid
                  :pictures     (db/images-by-user userid)}))

(defroutes gallery-routes
  (GET "/gallery/:userid" [userid]
       (display-gallery userid)))
```

你应该注意到了，通过我们将显示逻辑移动到模板中，应用程序的代码显著收缩。

剩下的任务就是创建用于用户注册和文件上传的模板。我们来看看如何实现它们。

转换注册页

在 auth 命名空间有两个负责显示注册和删除账户的页面。我们现在分别为这两个页面创建独立的模板。

注册页很直截了当，就是简单为用户输入 ID 和设置密码提供一个表单。我们扩展 base 布局，在内容块中添加这个表单。我们将其命名为 registration.html。注意，我们在此页为菜单提供一个空块。

picture-gallery-selmer/src/picture_gallery/views/templates/registration.html

```
{% extends "picture_gallery/views/templates/base.html" %}
{% block menu %}
{% endblock %}
{% block content %}
<div class="content">
    <form action="{{context}}/register" method="POST">
<label for="user-id">user id</label>
<input id="id"
       name="id"
       tabindex="1"
       type="text"
       value="{{id}}" /><br />
<div class="error">{{id-error}}</div>
<label for="pass">password</label>
<input id="pass"
       name="pass"
       tabindex="2"
       type="password" /><br />
<div class="error">{{pass-error}}</div>
<label for="pass1">retype password</label>
<input id="pass1"
       name="pass1"
       tabindex="3"
       type="password" /><br />
        <input tabindex="4" type="submit" value="create account" />
    </form>
</div>
{% endblock %}
```

除了我们需要给出关于删除的确认或放弃，账户删除页面看起来已非常类似了。我们将此文件命名为 deleteAccount.html。

picture-gallery-selmer/src/picture_gallery/views/templates/deleteAccount.html

```
{% extends "picture_gallery/views/templates/base.html" %}
{% block content %}
<div class="gallery">
    <form action="{{context}}/confirm-delete" method="POST">
        <input type="submit" value="delete account" />
    </form>
    <form action="{{context}}/" method="GET">
        <input type="submit" value="cancel" />
    </form>
</div>
{% endblock %}
```

与此同时，我们用于/register 和/delete-account 的路由 handler 简单显示各自的模板即可：

picture-gallery-selmer/src/picture_gallery/routes/auth.clj

```
(defn registration-page [& [id]]
  (layout/render "registration.html"
                 {:id id
                  :id-error (first (vali/get-errors :id))
                  :pass-error (first (vali/get-errors :pass))}))
```

转换上传页面

还是一样，我们先设计页面模板，再修改代码适应它。

picture-gallery-selmer/src/picture_gallery/views/templates/upload.html

```
{% extends "picture_gallery/views/templates/base.html" %}

{% block content %}
<div class="gallery">
      <h2>Upload an image</h2>
{% if error %}
<p class="error">{{error}}</p>
{% endif %}
{% if image %}
<img src={{image}} width="300" height="300"/>
{% endif %}

<form action="{{context}}/upload"
      enctype="multipart/form-data"
      method="POST">
   <input id="file"
          name="file"
          type="file" />
   <input type="submit" value="upload" />
      </form>
</div>
{% endblock %}
```

这里，我们创建上传图片的表单，并为上传操作的结果提供占位符。如果上传成功，则会给予缩略图的链接，否则会显示操作的错误内容。

我们现在对 upload 命名空间稍做修改，将模板用上：

picture-gallery-selmer/src/picture_gallery/routes/upload.clj

```
(defn upload-page [params]
  (layout/render "upload.html" params))

(defn handle-upload [{:keys [filename] :as file}]
  (upload-page
    (if (empty? filename)
      {:error "please select a file to upload"}
      (try
        (upload-file (gallery-path) file)
        (save-thumbnail file)
        (db/add-image (session/get :user) filename)
        {:image (thumb-uri (session/get :user) filename)}
        (catch Exception ex
          (error ex "an error has occured while uploading" name)
          {:error (str "error uploading file " (.getMessage ex))})))))
```

picture-gallery-selmer/src/picture_gallery/routes/upload.clj

```
(defroutes upload-routes
  (GET "/img/:user-id/:file-name" [user-id file-name]
       (serve-file user-id file-name))

  (GET "/upload" [info] (restricted (upload-page {:info info})))

  (POST "/upload" [file] (restricted (handle-upload file)))

  (POST "/delete" [names] (restricted (delete-images names))))
```

如你所见，我们仅通过少许工作就能切换模板引擎。其实 Hiccup 和 Selmer 都有各自的优势。

7.2 升级为 ClojureScript

到目前为止，我们仅在服务端使用 Clojure，在客户端仍然使用传统的 JavaScript 处理逻辑。ClojureScript 是一种编译为 JavaScript 的 Clojure 方言。这允许我们在客户端使用和服务端一致的语言。不仅如此，我们甚至还可以在前后端共享代码。

我们来看看为什么大家倾向于在后端开发中使用 ClojureScript。如果你使用 JavaScript 开发，就应该已经注意到它有一些缺点。我们来看看这些问题并见识一下 ClojureScript 怎么处理它们。

JavaScript 语法满是怪癖，虽然编写代码还算容易，看起来也像回事，但是事实上，完全不是这么回事。用等式检查这个例子来说明这个问题恰好不过了。由于

数据类型不同，等式判断规则也不一样，这就导致很容易被弄错，会弄出如下丑陋的代码：

```
if (typeof my_var !== "undefined" && my_var !== null) {
  // wow, that's ugly!
}
```

JavaScript 代码表达起来非常累赘，更不适合元编程。它需要花费大量不必要的气力来组织代码结构，甚至连内建命名空间都不支持。

库程序和开发小组在代码规范上面难以一致，还存在严重的争议。比如有些人写的是函数式风格的 JavaScript，有些人用此来编写面向对象的代码。这些约定基本无法统一。

在 JavaScript 上面，还没有能媲美 Leiningen 的工具。这意味着，你不得不手工跟踪依赖项和库程序。

因为这些原因，得花费很大的力气才能编写出健壮的 JavaScript 程序。尽管这样，JavaScript 有着能支持在所有现代浏览器中标准可编程环境的优势。由于在客户端已然成为标准，JavaScript 引擎的性能在近些年也得到突飞猛进的发展。

如果能借助健壮的语言（比如 Clojure）的平台优势该有多好。这就是 ClojureScript 展露拳脚的地方。

除了解决这些问题，ClojureScript 还有个优势是在客户和服务端使用同样的语言。这意味着我们可以在前后端共享逻辑，而不用重复编写代码导致引入错误或者内容不一致。

就像它的表亲 Clojure，ClojureScript 拥护其宿主平台，并且允许与 JavaScript 无缝交互。我们不仅可以使用成熟的 JavaScript 库，还可以享受 Clojure 这门语言带来的好处。

ClojureScript 概要

你应该意识到使用 ClojureScript 的几个关键。由于 ClojureScript 运行在浏览器中，我们再不能依赖项和利用任何 Java 代码，只有纯 Clojure 编写的库才可用。与 JavaScript 交互的代码和与 Java 交互略有不同。

JavaScript 交互

与 JavaScript 交互明显变得简单多了。任何标准 JavaScript 函数都可以通过 js 命名空间访问。举个例子，如果我们想生成日志并打印到控制台，可以写下如下代码：

```
(defn log [& items]
  (.log js/console (apply str items)))
```

与 JavaScript 对象属性交互并不是一件很突兀的事情。通过使用（.-property obj）就能与之交互，通过连字符（-）就能指明引用属性而不是使用函数。我们通过调用 set! 函数更新属性。这里有个例子：

```
(defn init []
  (let [canvas (.getElementById js/document "canvas")
        ctx    (.getContext canvas "2d")
        width  (.-width canvas)
        height (.-height canvas)]
    (log "width: " width " height: " height)
    ;;set a property
    (set! (.-fillStyle ctx) "black")
    (.fillRect ctx 0 0 width height)))
```

宏

另一个地方是 ClojureScript 和 Clojure 不同之处，就是在引用宏的时候，在命名空间定义处使用:require-macros 关键字。

```
(ns my.app
  (:require-macros [app.macros :as m]))
```

注意：as 在引用宏的时候是必不可少的。

并发

ClojureScript 现已支持 atom，暂时还不支持软事务内存以及 ref 和 agent，与 binding 的语义还有细微的区别。同时，它还没有在运行时可物化的 Vars。

最后，ClojureScript 还没有运行时求值或者编译。除了这些差异，ClojureScript 开发非常类似标准 Clojure。

我们将相册 JavaScript 部分移植到 ClojureScript。最简单的方式就是在项目中使用 lein-cljsbuild[①]来添加 ClojureScript 支持。你可以通过它来指定 ClojureScript 来源，最好引用 Clojure 命名空间，以及指定输出的 JavaScript 文件。

我们首先需要添加 ClojureScript 库以及 Leiningen 插件我们的到项目，添加如下代码到 project.clj：

```
:dependencies [...
               [org.clojure/tools.reader "0.7.10"]
               [org.clojure/clojurescript "0.0-1806"]]
:plugins [... [lein-cljsbuild "0.3.2"]]
:cljsbuild
{:builds
  [{:source-paths ["src-cljs"]
    :compiler
    {:pretty-print false
     :output-to "resources/public/js/gallery-cljs.js"}}]}
```

在:cljsbuild 下的配置指定 ClojureScript 代码会放置在 src-cljs 源码文件夹。ClojureScript 会被编译输出到 gallery-cljs.js，放在项目的 resources/public/js/目录中。

我们在项目的根目录创建 src-cljs 文件夹。我们对需要转换的 JavaScript 创建与之同名的命名空间，名为 gallery.cljs 和 site.cljs。

注意 ClojureScript 文件必须以.cljs 后缀结尾。如果你使用.clj 后缀，那么编译器依然试图编译命名空间，但是不能与 JavaScript 交互。这意味着你试图访问任何原生 JavaScript 函数都会报错。

使用 lein 调用 ClojureScript 编译器有两个可选参数，即运行 lein cljsbuild once 或 lein cljsbuild auto。当开启 auto 选项，生成处理会通过检查代码变化来自动选择是否需要重编译。这样做比使用 once 参数编译所花费的时间少得多。事实证明在开发中颇为便捷。

如果我们希望清理之前生成的产物，可以运行 lein cljsbuild clean。

我们使用 Domina[②]库来完成操作文档对象模型（DOM，Document-object Model）

① https://github.com/emezeske/lein-cljsbuild
② https://github.com/levand/domina

元素以及处理其事件，通过 cljs-ajax[①]处理 Ajax 请求。我们可以通过在 project.clj 中类似别的库包含方式加载这些库。

```
:dependencies [...
               [domina "1.0.0"]
               [cljs-ajax "0.2.0"]]
```

这时，我们已经可以通过运行如下命令启动 ClojureScript 编译：

```
lein cljsbuild auto
```

一旦运行自动构建，src-clj 源码目录中发生变化的所有文件都会被重编译。在浏览器中重新加载时就可以生效了。

首先，我们来看当前的 JavaScript，比如打开 site.sj。

```
picture-gallery-selmer/resources/public/js/site.js
function colorStr(color) {
    return "rgb("+color[0]+","+color[1]+","+color[2]+")";
}
function setColor(div, colors) {
    var bgColor = colors[0];
    var textColor = colors[1];
    div.css("background-color", colorStr(bgColor));
    div.find('a').css("color", colorStr(textColor));
}
$(document).ready(function(){
    $(".thumbnail")
            .each(function() {
             var div = $(this);
             var url = div.find('img').attr('src');
             var thumbColors = new AlbumColors(url);
            var color = "";
            thumbColors.getColors(function(colors) {
            setColor(div, colors);
        });
    });
});
```

这里，我们使用 AlbumColors 库提取缩略图色调，设置缩略图所在 div 的背景色和前景色。我们来看看在 ClojureScript 中怎么调用这个库。

① https://github.com/yogthos/cljs-ajax

picture-gallery-cljs/src-cljs/site.cljs

```
(ns site
  (:require [domina :refer [by-class nodes sel attr]]
            [domina.css :refer [sel]]))

(defn rgb-str [[r g b]]
  (str "rgb(" r "," g "," b ")"))

(defn set-color [style foreground background]
  (set! (.-color style) (rgb-str foreground))
  (set! (.-backgroundColor style) (rgb-str background)))

(defn img-url [div]
  (-> div (sel "img") (attr "src")))

(defn set-colors [div]
  (.getColors (js/AlbumColors. (img-url div))
    (fn [[background _ foreground]]
      (set-color (.-style div) foreground background))))

(defn ^:export init []
  (doseq [div (nodes (by-class "thumbnail"))]
    (set-colors div)))
```

如你所见，这段代码和 JavaScript 版本非常类似。它们主要的区别在于我们使用一个初始化函数注入（hook into）到 ClojureScript。注意，这里有个^:export 的标志。我们需要这个标志在高级编译过程中保护函数名。默认情况下，Closure 编译器会在目标代码中使用短名。

我们现在更新 base.html 模板访问被编译的 gallery-cljs.js 脚本。我们已经可以移除旧的 site.js、gallery.js 脚本以及 jQuery 的引用，它们现在已经不需要了。

```
<!DOCTYPE HTML>
<html>
    ...
        <script src="{{context}}/js/gallery-cljs.js" type="text/javascript">
        </script>
        <script>
            site.init();
        </script>
    </body>
</html>
```

注意，所有的ClojureScript命名空间会被编译输出到一个独立的JavaScript文件中。

相册网站的前后端都访问这同一个文件，仅需要被加载一次。

现在，我们来看看将 gallery.js 转换为 ClojureScript。这里，我们对 Ajax 的调用使用 cljs-ajax 库实现。

这个 gallery.js 如下所示。

picture-gallery-selmer/resources/public/js/gallery.js

```
$(document).ready(function(){
    $("#delete").click(deleteImages);
});
function deleteImages() {
    var selectedInputs = $("input:checked");
    var selectedIds = [];
    selectedInputs
    .each(function() {
                selectedIds.push($(this).attr('id'));
            });
    if (selectedIds.length < 1) alert("no images selected");
    else
        $.post(context + "/delete",
               {names: selectedIds},
               function(response) {
                  var errors = $('<ul>');
                  $.each(response, function() {
                  if("ok" === this.status) {
                  var element = document.getElementById(this.name);
                   $(element).parent().parent().remove();
               }
               else
                   errors
                   .append($('<li>',
                         {html: "failed to remove " +
                                this.name +
                                ": " +
                                this.status}));
                   });
                   if (errors.length > 0)
                       $('#error').empty().append(errors);
               },
              "json");
}
```

这段脚本基于服务端的回响，删除用户相册中的图片，并将图片上传到页面。我

们打开 gallery.cljs，更新命名空间定义处，添加必要的依赖库引用。

picture-gallery-cljs/src-cljs/gallery.cljs

```
(ns gallery
  (:require [goog.dom :as dom]
            [domina :refer [by-id nodes append!]]
            [domina.events :refer [listen!]]
            [domina.css :refer [sel]]
            [ajax.core :refer [POST]]))
```

由于仅希望在相册页面加载 gallery-related 函数，我们为此创建一个 init 初始化函数。这个函数会绑定删除按钮的点击事件。

```
(defn ^:export init []
 (listen! (by-id "delete") :click deleteImages))
```

我们现在需要在页面中调用 gallery.init()，而不是在 gallery-cljs.js 加载之后。我们可以在 base.html 创建一个新的脚本块，在这里添加那些脚本。

```
<html>
  <head>
  ...
  </head>
  <body>
...
  {% block scripts %}
  {% endblock %}
  </body>
</html>
```

我们这时在 gallery.html 中调用 init 脚本：

```
{% extends "picture_gallery/views/templates/base.html" %}
...
{% block scripts %}
<script>
  gallery.init();
</script>
{% endblock %}
```

我们来看看如何实现删除图像（deleteImages）函数。首先，我们使用 CSS 选择器从多选框收集选择结果。然后，我们通过查询结果获取结点，收集每个结点的 name 属性并返回。如果我们不能找到任何输入框，那应该返回 nil。

```
(defn find-selected []
  (->> (sel "input:checked")
       nodes
       (map #(.-name %))
       not-empty))
```

在 deleteImages 函数中，我们使用刚才编写的 find-selected 尝试搜寻选择框。如果选择为空，则显示错误内容。如果选择了图像，我们需要 POST 它们的名字到服务端。

picture-gallery-cljs/src-cljs/gallery.cljs

```
(defn deleteImages [_]
  (if-let [selected (find-selected)]
    (POST "/delete" {:params {:names selected}
                     :handler handle-response})
    (js/alert "no images selected")))
```

响应 handler 会检查每一项的状态，并从 DOM 中移除或是显示相关错误，这些情况与 JavaScript 版本一样。

picture-gallery-cljs/src-cljs/gallery.cljs

```
(defn handle-response [response]
  (let [errors (goog.string.StringBuffer. "")]
    (doseq [{:keys [name status]} response]
      (if (= "ok" status)
        (-> (by-id name)
            (.-parentNode)
            (.-parentNode)
            (dom/removeNode))
        (.append errors (str "<li>failed to remove " name ": " status "</li>"))))
    (let [error-str (str "<ul>" (.toString errors) "</ul>")]
      (if (not-empty error-str)
        (append! (by-id "error") error-str)))))
```

最终重构的 JavaScript 版本应该看起来是这个样子：

picture-gallery-cljs/src-cljs/gallery.cljs

```
(ns gallery
  (:require [goog.dom :as dom]
            [domina :refer [by-id nodes append!]]
            [domina.events :refer [listen!]]
            [domina.css :refer [sel]]
            [ajax.core :refer [POST]]))

(defn handle-response [response]
  (let [errors (goog.string.StringBuffer. "")]
```

```
(doseq [{:keys [name status]} response]
  (if (= "ok" status)
    (-> (by-id name)
        (.-parentNode)
        (.-parentNode)
        (dom/removeNode))
    (.append errors (str "<li>failed to remove " name ": " status "</li>"))))
(let [error-str (str "<ul>" (.toString errors) "</ul>")]
  (if (not-empty error-str)
    (append! (by-id "error") error-str))))
(defn find-selected []
  (->> (sel "input:checked")
       nodes
       (map #(.-name %))
       not-empty))

(defn deleteImages [_]
  (if-let [selected (find-selected)]
    (POST "/delete" {:params {:names selected}
                     :handler handle-response})
    (js/alert "no images selected")))
(defn ^:export init []
  (listen! (by-id "delete") :click deleteImages))
```

如你所见，使用 ClojureScript 实现客户端逻辑是件非常简单的事。

由于我们使用 ClojureScript 可以使用可扩展的数据符号[①]（EDN，Extensible Data Notation）格式将数据转换，用于客户端和服务端。这允许我们使用常规的 Clojure 数据结构，而无需编码为一个中间 JavaScript 对象表示。要使用它，我们需要引用 ring-middleware-format[②]的中间件来处理 EDN 编码请求。

```
:dependencies [...
               [ring-middleware-format "0.3.1"]]
```

cljs-ajax 库的 POST 函数正好将 EDN 作为默认格式。唯一需要做的，就是需要为 handler 添加中间件。我们首先在命名空间定义处添加它的引用。

```
(ns picture-gallery.handler
  (:require
   ...
    [ring.middleware.format :refer [wrap-restful-format]]))
```

① https://github.com/edn-format/edn
② https://github.com/ngrunwald/ring-middleware-format

然后简单地通过中间件应用程序定义：

picture-gallery-cljs/src/picture_gallery/handler.clj

```
(def app (noir-middleware/app-handler
          [auth-routes
           home-routes
           upload-routes
           gallery-routes
           app-routes]
          :middleware [wrap-restful-format]
          :access-rules [user-page]))
```

中间件会读出所有 EDN 编码的 body，并为请求设置为:params 键值。

到目前为止，我们已编译的 ClojureScript 未做任何优化。如果我们注意一下生成的脚本，会发现它大得惊人，接近 1MB！要是实际使用，这是绝对无法接受的。要想把尺寸降到接受范围，需要启动高级优化。

这次我们为开发和产品建立独立的配置，更新 project.clj 配置文件：

```
:cljsbuild
  {:builds
   {:dev {:source-paths ["src-cljs"]
          :compiler
          {:pretty-print true
           :output-to "resources/public/js/gallery-cljs.js"}}
    :prod {:source-paths ["src-cljs"]
           :compiler
           {:optimizations :advanced
            :output-to "resources/public/js/gallery-cljs.js"}}}}
```

为开发配置所输出的 JavaScript 更具有可读性，且更容易调试，所以没有必要优化。生产阶段的配置会压缩变量名，高级优化会尽可能压缩输出脚本的体积。

我们现在可以使用独立配置的名称作为 cljsbuild 的参数来编译：

```
cljsbuild:
lein cljsbuild once prod
```

尽管这样，当编译生产版本的 ClojureScript 后再打开页面，我们会看到如下错误提示：

```
Uncaught TypeError: Object [object Object] has no method 'Le'
```

很可惜，这个错误传达信息非常有限。这只是告诉我们访问的某个对象并不存在名为“Le”的方法。

这里的提示是告诉我们：我们从没有定义名为 Le 的方法。高级优化会压缩参数名。这并不是说函数定义有问题，因为我们为其命名一致。然而，我们还是能从 AlbumColors 库调用函数。编译器一样会压缩命名，明显还是找不到生成的函数名。

还好，这个问题解决起来很简单。GoogleClosure 编译器提供一种保护外部库函数名的机制，只要声明在一个外部文件即可。我们在项目的 resources 目录中新建一个名为 externs.js 的文件，在这里声明那些需要保护的函数：

```
picture-gallery-korma/resources/externs.js
var AlbumColors = {};
AlbumColors.getColors = function() {};
```

接下来，我们在:prod 构建配置中引用这个文件：

```
:prod {:source-paths ["src-cljs"]
       :compiler
       {:optimizations :advanced
        :externs ["resources/externs.js"]
        :output-to "resources/public/js/gallery-cljs.js"}}
```

现在，我们需要清理并全部重编 ClojureScript，进行如下操作：

```
lein cljsbuild clean && lein cljsbuild once prod
```

输出的脚本现在只有 150kB 左右。就我们的工作内容，这似乎比我们预想的要大，但你要知道，它还提供运行时的 ClojureScript 编译。这也不算什么太糟的事，加载类似 jQuery 的 JavaScript 库还用原生 JavaScript 代码呢！

7.3　SQL Korma

我们已经对前后端 Clojure 化了，现在，我们来着手对模型做同样的事。到目前为止，我们都是手工编写 SQL 查询语句，通过 clojure.java.jdbc 访问数据库。SQL Korma[①] 是个原生的 Clojure DSL 库，允许我们使用 Clojure 编写数据库操作。

① http://sqlkorma.com/

Korma 可以通过 Clojure 编写组合查询语句，并转换成 SQL 语句访问数据库。由于 DSL 简化了 SQL 结构，查询结果也变得高效且有可读性。首先我们需要在项目依赖项中添加 Korma：

```
[korma "0.3.0-RC5"]
```

我们现在可以移除对 org.clojure/java.jdbc 的依赖项，其实 Korma 也依赖它，实际上也会添加到项目中。要使用 Korma，我们仅需简单使用 defdb 封装数据库连接。

```
(defdb korma-db db)
```

我们使用 c3p0[①]库来创建数据库连接池并符合 db 规范。last-created 池默认用于所有的查询。

Korma 使用实体（entities）表示 SQL 表。实体代表查询的核心建立块，使用 defentity 宏来创建。我们将实体用作用户和图片表。这些实体名应与表名一致。

```
(defentity users)

(defentity images)
```

注意，只有一个连接指定的实体将被隐式地使用。我们现在可以重写数据库查询函数：

```
picture-gallery-korma/src/picture_gallery/models/db.clj
(defn create-user [user]
  (insert users (values user)))

(defn get-user [id]
  (first (select users
                 (where {:id id})
                 (limit 1))))

(defn delete-user [id]
  (delete users (where {:id id})))

(defn add-image [userid name]
  (transaction
    (if (empty? (select images
                        (where {:userid userid :name name})
                        (limit 1)))
      (insert images (values {:userid userid :name name}))
      (throw
```

① http://sourceforge.net/projects/c3p0/

```
      (Exception. "you have already uploaded an image with the same name")))))

(defn images-by-user [userid]
  (select images (where {:userid userid})))

(defn delete-image [userid name]
  (delete images (where {:userid userid :name name})))

(defn get-gallery-previews []
  (exec-raw
   ["select * from
   (select *, row_number() over (partition by userid) as row_number from images)
   as rows where row_number = 1" []]
   :results))
```

如你所见，Korma 风格的查询与标准 SQL 相当接近，同时也少了很多干扰。由于是使用原生 Clojure 编写，我们可以如同其他 Clojure 代码一样直接操纵。添加、选择、删除图片的函数都很容易移植：

picture-gallery-korma/src/picture_gallery/models/db.clj

```
(defn add-image [userid name]
  (transaction
    (if (empty? (select images
                  (where {:userid userid :name name})
                  (limit 1)))
      (insert images (values {:userid userid :name name}))
      (throw
        (Exception. "you have already uploaded an image with the same name")))))
(defn images-by-user [userid]
  (select images (where {:userid userid})))
(defn delete-image [userid name]
  (delete images (where {:userid userid :name name})))
```

到目前为止一切顺利，但是 get-gallery-previews 函数使用了复杂一点的查询，那么就不能简单通过 Korma 移植。这种情况下，我们可以使用 exec-raw 辅助函数直接使用查询语句，这类似于使用 clojure.java.jdbc 来实现：

picture-gallery-korma/src/picture_gallery/models/db.clj

```
(defn get-gallery-previews []
  (exec-raw
    ["select * from
     (select *, row_number() over (partition by userid) as row_number from images)
     as rows where row_number = 1" []]
     :results))
```

你可能注意到在默认情况下，Korma 会生成一些干扰的日志。要解决此问题，

我们需要加载 log4j 的依赖项，并为 Korma 添加日志配置。我们要添加如下依赖项到我们的项目：

```
[log4j "1.2.15"
  :exclusions [javax.mail/mail
               javax.jms/jms
               com.sun.jdmk/jmxtools
               com.sun.jmx/jmxri]]
```

这时，我们将 log4j.xml 配置文件放在项目的 resources 目录中。这个文件内容应如下：

```
picture-gallery-korma/resources/log4j.xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <logger name="com.mchange">
    <level value="WARN"/>
  </logger>
</log4j:configuration>
```

这样，Korma 就能处理 SQL 查询语句，并且可以使用标准的 Clojure 符号来表达逻辑了。

7.4 创建程序模板

一旦创建了一个明确类型的程序，比如相册应用，我们可能希望使用同样的结构再编写一个类似的应用。最好是创建一个程序框架模板，来完成这个任务。正好，我们可以使用 Leiningen 模板实现。

在本书中，我们都是使用 compojure-app[①]模板创建新项目。这里，我们再说说这些模板的工作机制并创建自己的模板。

Leiningen 使用名为 lein-newnew[②]的插件来创建模板。要创建一个新模板，我们仅需要运行 lein new template <template name>。

我们来探究一下 compojure-app 模板项目具体的工作机制。由于这是个 Leiningen 项目，它包含 project.clj。

① https://github.com/yogthos/compojure-template
② https://github.com/Raynes/lein-newnew

compojure-template/project.clj

```
(defproject compojure-app/lein-template "0.3.9"
:description "Compojure project template for Leiningen"
:url "https://github.com/yogthos/compojure-template"
:eval-in-leiningen true
:license {:name "Eclipse Public License"
          :url "http://www.eclipse.org/legal/epl-v10.html"}
:dependencies [[leinjacker "0.2.0"]])
```

这看起来是个普通的项目文件，除了多个 eval-in-leiningen 键，这是用来阻止 Leiningen 在编译期将给定的项目视为单独的进程。

模板本身是 src/compojure-template/leiningen/new/compojure_app.clj，内容如下：

compojure-template/src/leiningen/new/compojure_app.clj

```
(ns leiningen.new.compojure-app
  (:use [leiningen.new.templates :only [renderer sanitize year ->files]]
        [leinjacker.utils :only [lein-generation]]))

(defn check-lein-version []
  (if (< (lein-generation) 2)
    (throw (new Exception "Leiningen v2 is required..."))))

(defn compojure-app
  "Create a new Compojure project"
  [name]
  (check-lein-version)
  (let [data {:name name
              :sanitized (sanitize name)
              :year (year)}
        render #((renderer "compojure_app") % data)]
    (println "Generating a lovely new Compojure project named" (str name "..."))
    (->files data
             [".gitignore" (render "gitignore")]
             ["project.clj" (render "project.clj")]
             ["README.md" (render "README.md")]
             ["src/{{sanitized}}/repl.clj"          (render "repl.clj")]
             ["src/{{sanitized}}/handler.clj"       (render "handler.clj")]
             ["src/{{sanitized}}/routes/home.clj"   (render "home.clj")]
             ["src/{{sanitized}}/views/layout.clj" (render "layout.clj")]
             ["resources/public/css/screen.css"     (render "screen.css")]
            "resources/public/js"
            "resources/public/img"
            "src/{{sanitized}}/models"
             ["test/{{sanitized}}/test/handler.clj" (render "handler_test.clj")])))
```

这些神奇的功能是从 compojure-app 这个函数开始的。当我们运行 lein new compojure-app myapp 并使用这个模板来创建应用程序的时候，就是调用的这个函数。

模板中的代码基本都是自解释的。它使用 leiningen.new.templates/render 函数在指定路径创建模板文件。{{sanitized}}标签确保为包目录生成合法的名称（将——转换成下划线）。

我们发现模板文件本身都在 src/compojure-template/leiningen/new/compojure_app 路径。这些文件不需要和生成项目有一样的目录结构。你可以看看之前的代码，我们在模板中明确指定了目标路径。

在生成的项目中可以看出来，这些模板只是普通文件。唯一不同的是，当项目名被引用时，它们使用{{name}}锚。创建项目时指定的程序名会替换这个锚。我们拿 layout.clj 作为例子来看看：

compojure-template/src/leiningen/new/compojure_app/layout.clj

```
(ns {{name}}.views.layout
  (:require [hiccup.page :refer [html5 include-css]]))
(defn common [& body]
  (html5
   [:head
    [:title "Welcome to {{name}}"]
    (include-css "/css/screen.css")]
   [:body body]))
```

一旦创建了模板，我们就可以运行 lein install 将其安装到本地，然后可以开始使用它而不是写这类项目的样板。如果我们期望将其供于他人使用，可以通过运行 lein deploy clojars 将其发布到 Clojars。

7.5 你学到什么

在这一章，我们介绍了几种构建项目的办法。我们讨论了如何插入不同的模板引擎，还讨论了如何才能将 Clojure 用于服务端和客户端。最后还涉及如何使用 Clojure 语法通过 Korma DSL 来访问数据库。你大概注意到了，只要愿意，在程序中我们可以多用或者少用 Clojure。

最极端的情况是我们可以使用 Hiccup、ClojureScript 和 Korma。这种类型的程序几乎可以全部使用 Clojure 来完成。这种方法的主要好处是，我们可以将所有内

容统一使用一种语言实现。我们不需要记忆 SQL 或是 Javascript 的一些怪癖，甚至不用手写哪怕一条 HTML。这种方式最明显的缺点是恐怕只有精通 Clojure 的人才能驾驭。

另一个极端是我们使用 Selmer、JavaScript 和 clojure.java.jdbc 库。这里的 Clojure 只是用于实现程序的业务逻辑。这种方法将使用 Clojure 的规模降到最低。最重要的是，它允许我们和对这门语言不熟的人一起工作，比如 DBA 和 designer，而且还能轻易使用类似 AngularJs[①]强力的 JavaScript 框架。

使用 Clojure 实现多元化项目时，有一系列方案可提供灵活选择。最重要的是，这些方案还可以满足个人喜好，调整对 Clojure 的使用量。

Luminus 框架

Clojure 社区并不青睐传统风格的框架，而是偏爱可组合的库。问题是如何确定使用什么库？哪个库方便集成？哪个库容易维护？如果你是职业开发者，可能没有那么多时间来通过实践验证优劣。

我同意框架在传统意义上没增加多少价值，并认为对于做某件事情肯定还是存在一套精选库和基本流程的。这不仅容易上手，而且简化了遵循相同模式的其他项目的工作。

你可能还注意到在创建项目的时候生成了一些样板文件。你只需建立数据库、创建 handler、添加相关的中间件等即可。

Luminus[②]旨在为你处理这些事情，而你只需关注与编写程序的核心代码即可。

究其本质，Luminus 纯粹是个 Leiningen 模板，为典型的 Web 程序创建所有的样板文件。并不像多数模板，Luminus 为建立的项目配置使用可选参数表示具体的特性。比如，我们可以使用+postgres 来添加对 Postgres 数据库的支持，使用+cljs 引入对 ClojureScript 的支持。

如果我们要创建一个类似相册程序的应用，可以使用如下命令创建一个新项目：

```
lein new luminus picture-gallery +postgres +cljs
```

① http://angularjs.org/
② http://www.luminusweb.net/

这将创建一个配置使用 Postgres 数据库，包含 ClojureScript 勾挂。我们只需要将相关的代码添加到应用程序中即可。

结束语

这就是我们与 Clojure 开发 Web 应用程序的旋风之旅。我们广泛讨论了一系列的话题，包括开发工具、第三方库、配置以及部署。但愿你通过阅读本文，已经掌握进行进一步研究这些话题的未涉及部分（而你又迫切需要的话）的必要背景知识。

一些工具、类库的选择以及开发实践的讨论，反映了作者个人的经验以及当时构建 Web 应用的参数选择。在许多情况下，选择其他参数同样可行。同样，这里的目标是为你提供一个坚实的基础，这样才能通过自己去发掘。

Clojure 这门语言虽然还很年轻，但已有很多高质量的工具和库可用，以及一个无比奇妙的社区。我衷心希望你喜欢阅读本书，并通过书中内容启发你通过 Clojure 开发你的下一个 Web 应用。

附录 1

选择 IDE

在本书介绍了 Light Table 之后，你很可能也想试试别的编辑器，看看管理项目的效果如何。好在 Leiningen 项目对编辑器是透明的。在本附录内容中，我们来看看 Light Table 之外几个很流行的替代品。

安装 Eclipse

你可以从官网下载 Eclipse[①]，并选择为 Java 开发者专供的 Eclipse IDE 分支版本，它包含我们需要的一切。Eclipse 下载完成之后，马上解压并运行 Eclipse 的可执行程序吧。

Eclipse 通过 Counterclockwise[②]插件实现对 Leiningen 的全面支持。这允许我们在集成开发环境（IDE，Integrated Development Environment）中创建项目并管理依赖。

安装 Counterclockwise 插件

我鼓励你对 Clojure 启用 structural-editing（结构化编辑）模式（虽然这不是必需的）。在此模式中，编辑器会确保代码中的括号是成对的。你可能需要一点时间来适应它，但是我向你保证，这终将会极大提高你的工作效率。在菜单的高级选项中展开 Clojure-Editor，关闭 Strict/Paredit 编辑模式选项的 Start Editors。

配置 Eclipse

Eclipse 使用透视图的概念和视图显示为用户提供交互界面。每个透视图都为特定的任务做了优化，比如开发、调试、源码控制。在一个透视图内部，我们可以添加不同的视图，比如编辑器、项目浏览器、代码大纲等。

对 Clojure 开发最有用的透视图是 Java 透视图。我们可以添加包浏览、大纲、命名空间浏览以及控制台视图。如果这些没有显示，我们打开菜单中的 Window/Show View，勾选之前提到的视图，就可以显示了。

包浏览器允许我们浏览项目，打开其文件和依赖。大纲视图可以显示我们当前展开的命名空间所定义的所有函数。命名空间浏览器仅在 REPL 运行时被激活，它提供已加载的所有函数的预览。控制台允许我们查看来自程序写入标准输出和标准错误的内容。

安装 Emacs

Emacs 大概是现今活跃着的最老的 IDE 了。作为一个成熟的项目，它拥有大量的特性和插件可用，其中有着大量其他 IDE 所没有的同类功能。然而，这也意味着 Emacs 不遵循任何常见的模式，因为它诞生得比这些模式早太多了。

Emacs 能运行在各种主流的平台之上，可以从官方网站[①]下载。如果你使用 OS X，可能更需要为 Mac 定制的 Aquamacs[②]。

不像别的多数编辑器，Emacs 的快捷键由命令链组成。例如，如果想打开文件，首先输入 Ctrl-x，这会进入命令模式；然后输入 Ctrl-f，这时就允许搜索并打开文件。前面的命令连起来表示为 C-x - C-f，我们接下来马上就要用到这个命令。

其他的命令使用 meta（简写为 M）键调用。这个键通常在 Windows 和 Linux 系统

① http://www.gnu.org/software/emacs/
② http://aquamacs.org/

中是 Alt 键，在 OS X 上是 cmd。比如，我们可以通过运行 M-x package-refresh-contents 更新可用包。

一旦你下载并运行 Emacs，就需要对其进行配置，使其支持 Clojure 开发，这需要安装一些插件包。推荐在~/.emacs.d/目录下创建 init.el。我们现在为其添加初始化脚本。用这种方式有一大好处：当切换到另外一台机器的时候，仅需简单拷贝覆盖 init 文件，而不用重新配置 Emacs。

我们首先需要通过包仓库的 URL 初始化包管理。现在我们运行 C-x - C-f 指定 ~/.emacs.d/init.el 新建一个文件。

接下来，我们添加如下代码，来初始化包管理：

```
(require 'package)
(add-to-list 'package-archives
             '("marmalade" . "http://marmalade-repo.org/packages/"))
(package-initialize)
```

运行 C-x - C-s 保存文件。当文件保存之后，我们通过运行 M-x package-refresh-contents 刷新包列表。完成之后，包列表应该是最新的了。我们现在可以指定需要安装包，将以下代码添加入 init 文件：

```
(defvar my-packages '(clojure-mode
                      clojure-test-mode
                      nrepl))

(dolist (p my-packages)
  (when (not (package-installed-p p))
    (package-install p)))
```

这里，我们使用 clojure-mode 包[①]添加 Clojure 编辑支持。nREPL 包[②]提供用于网络 REPL 服务的 Emacs 客户端。这允许我们从 Emacs 连接 Leiningen 项目。

我们再次保存，通过运行 M-x eval-buffer 对缓存求值。新包就被下载并安装了。你可能看到过程中会出现一些警告，不用担心，无视就行了。

我们来看看如何启动 REPL 会话加载留言簿应用，通过以下步骤实现。

1．通过运行 C-x - C-f 指定 guest- book/src/guestbook/repl.clj 打开项目的命名空间。

① https://github.com/clojure-emacs/clojure-mode
② https://github.com/clojure-emacs/nrepl.el

2．选择了 repl.clj 之后，运行 M-x nrepl-jack-in 为项目启动 REPL 服务并连接。

3．运行 C-c - M-n 将 REPL 命名空间切换为当前缓冲的命名空间。

4．最后，运行 C-c - C-k，对当前缓冲求值。

如果以上都顺利执行，Emacs 已经成功连接项目，可以开始在上面干活了。最后，我推荐看看 Emacs Prelude[①]这个项目，对默认 Emacs 能力进一步强化。

替代品

如果你对用 Eclipse 和 Emacs 作为 Clojure 编辑器都不满意。那就再看看以下替代品。

Cursive

Cursive[②]是另一个不错的选择。这是个基于 IntelliJ 平台的新 IDE。编写本书时它还是 beta 版，不过已经用在很多项目中了，假如你现在就是 IntelliJ 用户，不容错过。它提供完整的结构编辑、Leiningen 依赖管理，通过 nREPL 支持 REPL。安装说明可以在其官方网站看到。

VimClojure

如果你是 Vim 用户，你可能想看看 VimClojure[③]。其插件支持很多预期特性，比如字符高亮、排版缩进、代码自动完成。VimClojure 最大的缺点是不支持结构化编辑。

① https://github.com/bbatsov/prelude
② http://cursiveclojure.com
③ https://bitbucket.org/kotarak/vimclojure

附录 2

Clojure 入门

关于学习 Clojure 的书籍，市面上已经有很多了。这里只做简单概述，假使你现在对 Clojure 还不熟，我希望你通过阅读本章，可以轻松阅读书中绝大部分代码。在讲解深度上，这里不只是简单介绍语法，我希望通过对 Clojure 这门语言简短、细致的讲解，让你领悟其独特之处。

乍看 Clojure 代码，你可能会觉得如天书一般，认为学起来定比其他语言要难得多。我可以向你保证，要弄懂 Clojure 压根就不是困难的事。通过简单实践，你甚至还会觉得简单、亲切。

大多数主流语言的主要差异基本上都是在语法糖（Syntactic Sugar）上，在不同语言之间过渡大都不怎么费力，不过碰到 Clojure 就不是那么回事了。作为一门 Lisp 方言，它来源于完全不同的语系，需要对此多多练习方能适应。如果开始阅读时觉得很费力也不用气馁，因为它有别于你记忆中的其他任何语法，这是建立一个全新的概念。

函数式理念

Clojure 是门函数式编程语言。这使其与现代程序的编写要求极为吻合。随着程序规模的扩增，人们希望能够对程序模块化隔离，这也需要对代码的可测试和可复用提出了更高的要求。

状态维护

函数式语言之所以成为编写大型程序的理想选择，很大程度上由于其天生就避开了全局状态以及推崇不变性。当数据以不变性为主导，我们就可以轻松隔离构思并编写模块化程序。

数据不可变乍一听像是个无厘头的想法。然而，很多函数式语言带来的优势都是基于此，并有着密不可分的联系。那我们就来看看不变性的数据结构如何成为强有力的工具。

在多数语言中，数据可以传递给变量或者引用。将数据传递给变量是相对安全的，因为对变量进行任何赋值，对数据所做的任何变更都是受控的，不会扩散到当前函数以外。然而，在有些情况下，这也会付出巨大的开销，所以对大数据量的操作大都使用传递引用来实现。这致使代码关系难以推测，你必须保障引用这份数据的所有地方都是安全的才行。

而不可变数据给予我们第三种选择。每次对数据的修改实际是创建一个新的版本。而为此付出的开销取决于我们修改内容的多少。当一块数据不再被引用，则会被垃圾回收机制自动释放。

我个人是喜欢使用垃圾回收而不是手动内存管理的。这允许我们随时修改“拷贝”数据，而不用担心数据从哪里来，或者考虑修改范围等问题。语言会为我们考虑哪些数据是再也不用的、何时需要被清理。

这种数据结构有助于我们编写纯函数（Pure Functions）。纯函数是个纯粹的函数，无副作用。给定同样的参数，函数就一定会输出同样的结果。由于这样的函数在模块隔离时可以被推断，所以很容易将一个大型项目拆分成若干彻底独立的模块单元编写。这种类型的代码称为引用透明。

实现代码复用

面向对象语言往往对数据和相关操作有着极强的耦合。在这种情况中，要重用写在类里的方法并不是件简单的事，遇到类似的问题需要处理时，我们不得不重新编写。

由于逻辑和数据在函数式语言中本身就是分离的，这种问题压根就不存在。语言内建了一套常用且简洁的数据结构，类似 list（列表）、map（映射表）以及 set（集合表）。当面临一个新问题，我们可以复用所有针对数据结构操作的函数。

一个函数代表一个具体的转换，我们可以随意操作数据。当遇到需要解决的问题时，我们可以将其理解为序列的变换，并映射为适当的函数操作。这种代码风格称为声明式。

声明式代码将做什么和如何做分离。当我们需要对一个集合执行迭代，使用迭代函数即可，而迭代执行的逻辑作为参数传递。

这种风格最大的优势是可以在函数级别复用代码。一个迭代函数可以只用写一次边缘处理以及边界检查。然后我们可以复用此逻辑，而不用每次都操心检查的事。

向并发借力

函数式编程解决并行和并发这种难题也是手到擒来。对这种问题倒不是真有灵丹妙药，而是语言提供了解决办法让处理容易许多。

由于纯函数仅依赖其参数，并不借助共享状态，所以可以安全实现并行。这意味着我们可以轻易实现并行计算，充分发挥多核的优势。比如对集合的 map 操作就很能说明问题。我们可以写下一个使用 map 函数的版本。你应该发现每次操作会耗费大量的时间，简单使用并行版 pmap 替换掉再看看？

除此之外，不变性的数据结构也提供了卓越的工具来管理共享状态。Clojure 提供基于数据结构的软事务内存（STM，Software Transactional Memory）库。有了事务内存，在多线程访问共享数据时，再也不用担心出错而手工上锁。此外，其实数据仅在写入时才需要上锁。由于现有数据拥有不变性，即使更新时也可以安全读取。

数据类型

Clojure 提供丰富的数据类型，绝大多数都是耳熟能详的。

- Vars 提供本地存储变量。它允许基于独立线程的绑定和重绑定。
- Booleans 的值为 true 或 false，nil 会被视为 false。
- 数字有整数、双精、浮点还有分数。

- 符号（Symbol）用来作为变量的标示符。
- 关键字（Keyword）是引用自身的符号，前缀为冒号（:），常用于 map 的键名。
- 字符串通过双引号表示，允许跨行。
- 字符通过正斜杠前缀表示。
- 正则表达式是前缀为井号的字符串。

除了以上基本的数据类型外，Clojure 还提供一组功能丰富的标准集合类型。其包含 List、Vector、Map 和 Set。

- List: (1 2 3)
- Vector: [1 2 3]
- Map: `{:foo "a" :bar "b"}`
- Set: `#{"a" "b" "c"}`

有趣的是，你刚看到的是 Clojure 使用数据结构写的代码。这和绝大多数语言不同，你可以使用同样的符号来定义数据以及编写问题逻辑。使用同样的符号还能实现强大的元编程特性。

你可以用操纵数据结构的方式来编写 Clojure 代码。在创建模板中针对问题域常遇到的片段重复，这里提供一条便捷的途径去解决。

使用函数

函数调用在 Clojure 中非常类似于别的语言。我们来对比看看 Python 和 Clojure 都如何调用函数：

```
functionName(param1, param2)

(function-name param1 param2)
```

在 Clojure 版本中，最大的区别就是函数名在括号里面。这是因为函数调用其实是包含函数名和其参数的列表（list）。列表在 Clojure 中是种特殊的数据结构，编译器会认为列表的第一个元素是可调用的。如果你想创建一个列表数据结构，你应该调用 list 函数。

```
(list 1 2 3)
```

匿名函数

Clojure 允许创建没有函数名的函数。这种函数被称为拉姆达表达式（lambda expressions），使用 fn 表达式定义。这种特殊格式实质是个原语表达式，有别于标准表达的规则。每个特殊表达式对参数进行唯一求值。

举个例子，假如我们需要一个通过参数打印输出的函数，我们可以如下编写：

```
(fn [arg] (println arg))
```

我们可以传给它一个参数来输出：

```
((fn [arg] (println arg)) "hello")
=>"hello"
```

Clojure 还提供一个语法糖，使用#定义匿名函数。前面这个函数还可以通过以下代码简化重写：

```
#(println %)
```

这里%表示匿名参数，如果函数接受多个参数，其后再跟随一个数字来说明参数位置。在下一个例子可以看到：

```
#(println %1 %2 %3)
```

上面这个匿名函数接受三个参数，并将其按顺序打印输出。这种函数用在需要执行一次性操作的地方非常合适，而不用去正式地定义一个命名函数。这种函数通常配合高阶函数(Higher-order Function)使用，我们待会再讲。

命名函数

命名函数实质是将一个匿名函数绑定到一个符号上，通过一个标识符访问。Clojure 提供一个叫作 def 的特殊表达式来创建全局变量。其接受一个标识名和一个分配的主体，我们使用 def 创建一个命名函数：

```
(def double (fn ([x] (* 2 x))))
```

由于这是个常规操作，Clojure 为我们提供名为 defn 的宏来实现：

```
(defn square [x]
  (* x x))
```

defn 的第一个参数是定义的函数名。其后是包含参数的 vector 以及函数体。在前面的代码，我们仅传递一个参数给函数体。不过，我们可以这样传递多个参数：

```
(defn bar [a b]
  (println a)
  (println b)
  (* 2 (+ a b)))
```

这里，我们定义的函数 bar 接受 a 和 b 两个参数。函数体由两行打印，ab 求和再乘以 2 的返回组成。在最后一行表达式中，先对(+ ab)求值，接下来将其结果乘以 2 并作为 bar 函数的返回值。

需要注意的是，Clojure 使用单次扫描编译（Single Pass Compiler）。由于这个原因，函数必须在使用之前定义。实际使用中，有时我们需要在函数定义之前调用。我们使用 declare 宏来对函数进行前置声明。

```
(declare down)

(defn up [n]
  (if (< n 10)
    (down (+ 2 n))
    n))
(defn down [n]
  (up (dec n)))
```

你可能还注意到，代码是树形结构。这个树形结构被称为抽象语法树（AST，Abstract Syntax Tree）。此处的 AST 和编译时编译器使用的是一致的。通过直接看到的 AST，我们可以直观了解模块之间的逻辑关系。

通过对数据处理的代码，我们能看出来，它的语法要比绝大多数语言都要简练。例如，你可能也注意到了，在函数体中没有明确的返回语句，而是隐式地对最后一个表达式求值并将其作为返回值。如果你所接触的绝大多数语言都有明确的表达，那么这可能需要花点时间来适应。为了保持可读性，原则上函数体不超过五行，同时，缩进和空行都可以将代码划分开来，一目了然。

Clojure 中，函数和变量没有本质区别。你可以为函数分配名称，将其作为参数传递，也可以作为函数返回值返回。函数可以如同数据一般被作为“一等公民”对待，因为具体使用上没有任何的附加限制。

高阶函数

一个接受函数作为参数的函数被称为高阶函数。比如 map 就是这样一个函数：

```
(map #(* % %) [1 2 3 4 5])
=>(1 4 9 16 25)
```

这里我们将两个参数传给 map。第一个参数是一个对参数执行平方的匿名函数，第二个参数是一个数字集合。map 函数会访问集合的每一个元素并求平方。使用高阶函数的一大优点是，我们不用操心边界情况，比如空检查，迭代函数会为我们处理这些事情。

另外一个高阶函数的例子是 filter。这个函数遍历整个集合，只保留符合条件的元素。

```
(filter even? [1 2 3 4 5])
=>(2 4)
```

当然，你还可以将这些函数连在一起用解决问题。

```
(filter even?
  (map #(* 3 %) [1 2 3 4 5]))
=>(6 12)
```

这里我们先对每个元素乘 3。然后使用 filter 将结果序列中的偶数提取出来。

有了高阶函数，实际上你彻底不用再写循环或者递归代码了。当你需要迭代一个集合，使用类似 map 或者 filter 函数就行了。加之 Clojure 提供了丰富的标准库，任何数据变换都可以通过多个高阶函数组合实现。

你不用再学习不同语言的特性、语法规范，只用简单学习标准库的函数即可。一旦你掌握了通过特殊函数处理相关数据转换，就可以通过简单将已有函数组合解决很多问题。

这里有个真实的案例。问题是如何通过给定的多个地址字段显示格式化地址。通常，地址由国家、邮编、城市、街道、门牌号组成，我们会检查每一个字段，移除无效或空字段，再在中间插入分隔符。

假设我们在数据库中有包含如下字段的数据表：

```
unit        | street          | city     | postal_code | country
""          | "1 Main street" | Toronto | nil          | Canada
```

假定前面给定的数据是字符串，我们需要如下格式化输出字符串：

```
1 Main street, Toronto, Canada
```

现在，我们的首要任务是找到一个用来移除空字段的函数，然后在中间插入分隔符，最后连接成一个字符串输出：

```
(defn concat-fields [& fields]
  (apply str (interpose ", " (remove empty? fields))))

(concat-fields "" "1 Main street" "Toronto" nil "Canada")
=>"1 Main street, Toronto, Canada"
```

注意，我们在写代码的时候，还没有指示如何去完成这项任务。多数时候，我们所做的仅仅是组合需要的代表操作的函数。

闭包

接下来，我们来看看返回函数的函数。在面向对象语言中，这种函数是通过构造函数实现并体现出来的。例如，如果希望通过函数来初始化一些变量，你可以如下实现：

```
(defn make-client [url]
  (fn [request] (str "sending " request " to " url)))
(let [client (make-client "http://foo.org")]
  (println (client "request 1"))
  (println (client "request 2")))
```

这里，我们创建一个函数，接受 url 参数并返回一个函数，返回的这个函数接受请求作为参数。内部函数可以访问 url 变量，因为这个变量定义在它的访问范围内。

这种函数称为闭包（Closure），因为它将参数封闭。这里是 url，我们将其用在返回函数中。

你应该注意到，我们通过使用 let 表达式绑定 client 符号。let 表达式可以存放用到的任何表达式，在命令式语言中是通过定义变量的形式体现的。

流表达式

你现在可能会疑惑，如果表达式嵌套过多，是否会降低可读性。还好，Clojure 针对这个问题为我们提供了两个辅助表达式。假设有一组十个数字的范围序列，我们想将其每个数字加 1，然后为序列的元素之间插入数字 5，最后对全序列求和并返回。我们可以如下编写：

```
(reduce + (interpose 5 (map inc (range 10))))
```

乍一看，很难说出这个表达式中首先执行哪一个。如果每个步骤再复杂一点，就更没法看了。另外，假如我们需要对执行顺序进行调整，比如将加 1 和插入 5 两步调换，估计得重新将顺序捋一遍了。

或者，我们可以使用->>表达式来重写上面的语句：

```
(->> (range 10) (map inc) (interpose 5) (reduce +))
```

这里我们使用->>将一个个操作表达式贯穿起来。这意味着，我们将每个表达式的结果传递给下一个表达式，直到最后一个表达式输出。如果我们需要将结果传递到第一个参数，我们可以使用->表达式。

惰性化

许多 Clojure 求值都是惰性的。换句话说，直到结果真实需要、被具体使用，求值表达式才真正执行。这对于很多算法在效率上至关重要。

例如，你可能会认为我们前面这个例子非常低效，因为创建序列之后，每次都要对其迭代、插入、归约。

然而事实并不是这样。只有在需要的时候才会对表达式求值。从第一个值，也就是序列创建之后，一个接一个地传递给后面的函数，直到这个队列结束。这种方式很类似 Python 中的迭代器。

结构化代码

Clojure 和绝大部分语言的非常重要的区别在于代码结构上。在命令语言的常规做

法中，不同行的代码为了修改共享数据。每一个行都能访问前面代码行的结果。

例如，如果我们有一个整数列表，并且希望去对其每一个元素求平方，并打印其中的偶数，下面是 Python 代码并且是完全可用的：

```
l = [1, 2, 3, 4, 5]
for i in l
  i = i*i
for i in l
  if (i mod 2 == 0)
print l
```

在 Clojure 中，这种交互很明显，不是创建一个共享储存单元，而是使用不同函数接连访问它。我们将函数连在一起，最终通过它们输出：

```
(println
  (filter #(= (mod % 2) 0)
    (map #(* % %) (range 1 6))))
```

或者，如前面讲的，我们可以使用->>宏将操作扁平化：

```
(->> (range 1 6)
     (map #(* % %))
     (filter #(= (mod % 2) 0))
     (println))
```

每个函数都返回一个新的值，而不是对已有的数据进行修改。你可能会认为这样会付出巨大开销，并且认为它只是单纯地在每次数据修改时都对数据进行全拷贝。

实际上，Clojure 背后是可持久化的数据结构（Persistent Data Structures）[①]，创建数据的内存修正。每次的变更会导致创建一个新的修订，这个修订与修改的尺寸成正比。通过这种方式，我们只用付出介于旧数据和新数据结构差异之间的代价，同时确保任何变更都是局部的。

非结构化数据

Clojure 有个非常厉害的机制，在数据结构中称之为对声明访问的解构。如果知道数据的具体结构，你可以在绑定时通过文字符号描述。我们通过一些例子来看看具体

① http://en.wikipedia.org/wiki/Persistent_data_structure

含义：

```
(let [[smaller bigger] (split-with #(< % 5) (range 10))]
  (println smaller bigger))

=>(0 1 2 3 4) (5 6 7 8 9)
```

这里，我们使用 split-with 将 10 个数字的序列分割成两个部分：数字小于 5 和大于等于 5。由于知道结果格式，我们可以将其写成字面形式：[smaller bigger]，其中的两个命名元素会在 let 范围内实现值绑定。

我们同样可以将解构用在函数定义时。接下来，我们有个叫作 print-user 的函数，它接受一个 vector 的参数，其中有三个元素，名称分别为 name、address、phone。

```
(defn print-user [[name address phone]]
  (println name address phone))

(print-user ["Bob" "12 Jarvis street, Toronto" "416-987-3417"])
```

我们还可以指定变量参数，将其作为一个序列，那样就能支持不同参数的个数。通过使用&后面跟一个参数列表名来实现。

```
(defn foo [& args]
  (println args))

(foo "a" "b" "c")
=>(a b c)
```

由于变长参数保存在序列中，我们同样可以对其使用结构。

```
(defn foo [first-arg & [second-arg]]
  (println (if second-arg
             "two arguments were passed in"
             "one argument was passed in")))

(foo "bar")
=>"one argument was passed in"
(foo "bar" "baz")
=>"two arguments were passed in"
```

我们还可以对 map 使用解构。在解构 map 时，我们创建本地名字绑定的 map，用来从原始 map 中提取对应键：

```
(let [{foo :foo bar :bar} {:foo "foo" :bar "bar"}]
  (println foo bar)
```

数据结构可能是嵌套的，但一样也能解构。只要知道具体的数据结构，你都可以简单写下来：

```
(let [{[a b c] :items id :id} {:id "foo" :items [1 2 3]}]
  (println id " has the following items " a b c))
```

最后，作为常用操作，这里还有一些提取 map 键的语法糖。例如，如果有个 map 包含键:id 和:password，我们可以写一个 login 函数，在定义时提取这些键。

```
(defn login [{:keys [id password]}]
 ...)
(login {:id "bob" :password "secret"})
```

我们经常只想提取部分键，但也希望能用原始 map。

```
(defn [{:keys [id pass pass1] :as user}]
  (if (and id (= pass pass1))
      (println "valid user")
      (println user " is not filled in correctly")))
```

命名空间

在实际应用当中，我们需要某种工具通过逻辑区分来组织代码。在面向对象语言中，它通常使用类，通过定义类成员方法的方式实现。而在 Clojure 中，我们将函数分属到命名空间。我们来看看命名空间如何定义。

```
(ns myns)

(defn print-message [message]
  (println "message:" message))

(defn say-hello [user]
  (print-message (str "hello " user))
```

这里有个名为 myns 的命名空间，包含两个函数：print-message 和 say-hello。在同一命名空间下，函数可以直接互访。不过，如果我们希望访问别的命名空间下的函数，则需要在 myns 命名空间定义处添加对那个命名空间的引用。

在 Clojure 中有两种方式引用命名空间。

:use 关键字

第一种方法是在定义时使用:use 关键字引用命名空间。当使用这种方式引用时，被引用的命名空间下的所有 Vars 隐式可用。

```
(ns myotherns
  (:use myns))
(say-hello "Bob")
```

这种方式也有个缺点，即使用函数时，我们不知道函数原始定义在哪里，这为浏览代码增加了负担，并且如果引用的两个命名空间有同名函数，还会出错。

对第一个问题，我们可以在:use 定义处使用:only 明确描述需要的函数，实现选择性引用。

```
(ns myotherns
 (:use [myns :only [say-hello]]))

(defn print-message [message]
  (println "in myotherns"))

(say-hello)
```

这种方式中我们描述清楚了 say-hello 来自何处，并且我们还能在 myotherns 命名空间定义自己的 print-message 而不至于冲突。无论如何，say-hello 调用的还是 myns 命名空间定义的 print-message。

:require 关键字

第二种方式是使用:require 关键字引用命名空间。:require 关键字允许使用多种策略，我们分别看看。

我们可以直接指明需要一个命名空间，而没有进一步指示。这种情况下，任何调用都需要在符号前加上命名空间名的前缀来指定来源。

```
(ns myotherns
 (:require myns))

(myns/say-hello)
```

这种方式明确了符号的来源，并能在引用多个命名空间时确保不会出现冲突。唯一的麻烦就是如果命名空间声明得太长，我们每次使用函数时都要写下长长的声明。为缓解这个问题，:require 声明提供了:as 指令，允许我们创建命名空间别名。

```
(ns myotherns
 (:require [myns :as m]))

(m/say-hello)
```

我们还可以在使用:require 引用命名空间的函数时使用:refer 关键字，这和之前看到的:use 起到的作用一样。

要从另外命名空间导入所有函数，我们可以如下表示：

```
(ns myotherns
 (:require [myns :refer :all]))
```

如果需要通过函数名选择性引用，我们可以这样写：

```
(ns myotherns
 (:require [myns :refer [say-hello]])
```

如你所见，有多种方法引用另一个命名空间的 Vars。如果你拿不定主意，推荐使用 require 引用命名空间，通过名字或别名访问，这样有利于理清路由关系。

动态变量

Clojure 提供支持声明动态变量，允许在特定范围内对变量值进行修改。我们来看看如何使用。

```
(declare ^{:dynamic true} *foo*)
(println *foo*)
=>#<Unbound Unbound: #'bar/*foo*>
```

这里，我们声明*foo*是个动态变量，还未对其进行赋值。当试图打印*foo*时，我们会被告知出现错误声明，这个变量还未绑定任何值。

我们来看看，如何使用 binding 为*foo*分配值。

```
(defn with-foo [f]
  (binding [*foo* "foo"]
    (f)))
```

```
(with-foo #(println *foo*))
=>foo
```

我们通过 with-foo 函数，将字符串“foo”赋给*foo*。当传入 with-foo 的匿名函数被执行之后，再试图打印其值，就不会得到错误。

这种技术用在处理资源时非常有用，例如文件流、数据库连接或是局部变量。一般来说，并不鼓励使用动态变量，因为它会使得代码不透明，并且很难推断。尽管这样，为了合理使用，有必要了解其使用方法。

召唤 Java

使用 Clojure 还有个优势，就是能依靠现有 Java 库丰富的生态系统。如果原生的有些功能不足，还可以通过调用 Java 库来实现特殊任务。调用 Java 类也非常容易，以下展示的标准 Clojure 语法与之非常接近。

引入类

当需要使用 Clojure 库，我们使用:use 和:require 声明。不过，当希望导入 Java 类，我们必须使用:import 声明。

```
(ns myns
 (:import java.io.File))
```

我们同样可以通过一个 import 从同一个包导入多个类，如下所示。

```
(ns myns
 (:import [java.io File FileInputStream FileOutputStream]))
```

实例化类

通过类创建实例，我们可以调用 new 操作，这跟在 Java 中操作一模一样。

```
(new File ".")
```

通常，我们还可以更便捷地创建一个新对象：

```
(File. ".")
```

调用方法

当对类进行实例化之后，我们即可调用其方法。形式上类似于常规的函数调用。当调用方法的时候，我们将对象的方法放在对象前面，随后是方法所接受的其他参数。

```
(let [f (File. ".")]
  (println (.getAbsolutePath f)))
```

这里，我们创建一个新的文件对象 f，并且调用了它的.getAbsolutePath 方法。注意方法前面有个点（.），这有别于常规 Clojure 函数调用。如果希望调用静态函数或是类变量，我们使用斜杠（/）符号，如下所示。

```
(str File/separator "foo" File/separator "bar")

(Math/sqrt 256)
```

在这里，一样也有针对多个方法一起调用连续调用的速记符“..”。假设希望获取文件路径的字符串并提取其字节块，我们可以有两种不同写法。

```
(.getBytes (.getAbsolutePath (File. ".")))

(.. (File. ".") getAbsolutePath getBytes)
```

第二种写法看起来更自然，并且更易读。虽然还有其他处理 Java 的语法糖，但我们这里不再做过多介绍，以上内容足以理解本书的有关材料。

动态多态

protocol 允许定义一个抽象的函数集，可以通过具体类型实现。我们现在来看看 protocol 的例子：

```
(defprotocol Foo
  "Foo doc string"
  (bar [this b] "bar doc string")
  (baz [this] [this b] "baz doc string"))
```

如你所见，Foo protocol 指定两个方法，bar 和 baz。方法的第一个参数是对象本身，后一个是其参数。注意 baz 方法有多种参数版本。我们现在可以创建一个类型并实现

Foo protocol，使用 deftype 宏：

```
(deftype Bar [data]
  Foo
  (bar [this param] (println data param))
  (baz [this] (println (class this)))
  (baz [this param] (println param)))
```

我们创建 Bar 类型实现了 protocol Foo。每个方法会打印输出它们的一些参数，我们来看看当创建一个 Bar 的实例，并调用它的一些方法会是什么样子：

```
(let [b (Bar. "some data")]
  (.bar b "param")
  (.baz b)
  (.baz b "baz with param"))

some data param
Bar
baz with param
```

第一个方法调用打印 data，这是在 Bar 初始化的时候传入的参数。第二个方法调用了打印对象的类。最后一个方法说明成功调用另一个 baz。

我们可以通过 protocol 来扩展已有类型的功能，包括现有的 Java 类。例如，我们可以使用 extend-protocol 扩展 java.lang.String 类 Foo protocol：

```
(extend-protocol Foo
  String
  (bar [this param] (println this param)))
(bar "hello" "world")

hello world
```

用来理解本书前面的例子，这里已经讲得够多了。不过，针对 protocol，还有一些别的用法，这里不再一一讲解，我鼓励大家花点时间自己去摸索。

全局状态怎么样

虽然主要是不变性，Clojure 还是通过 STM 库[①]提供支持共享可变数据。STM 确保对可变量的所有的更新都是原子操作。其主要有两类可变类型：atom 和 ref。atom

① http://clojure.org/concurrent_programming

用于处理需要做非协同更新，ref用在可能会出现并发修改的事务中。

我们通过例子来看看如何定义并使用atom。

```
(def global-val (atom nil))
```

我们定义了一个名为global-val的atom，当前值为nil。我们可以通过使用dref函数读取它的值，这样能返回当前值。

```
(println (deref global-val))
=>nil
```

由于这是个常用操作，这又有个针对 deref 的速写符号“@”。因此，(println @global-val)与前面的代码等价。

有两种方式为atom赋新值。我们可以通过使用reset!赋值，也可以使用swap!传入一个函数来根据当前值来更新。

```
(reset! global-val 10)
(println @global-val)
=>10
(swap! global-val inc)
(println @global-val)
=>11
```

注意swap!和reset!都是以“!”结尾，这是个约定惯例，用来指明本函数是对可变数据的操作。

定义ref和我们定义atom的方式一样，但是这两者的用法截然不同。我们快速浏览其使用方式。

```
(def names (ref []))

(dosync
(ref-set names ["John"])
(alter names #(if (not-empty %)
                (conj % "Jane") %)))
```

以上代码中，我们定义名为names的ref。接下来使用dosync声明打开一个事务。在事务中，我们将一个包含“John”的vector赋给names。接下来，我们调用alter去检查如果names不是空则添加“Jane”进入vector，否则保持不变。

注意，由于发生在事务中，空检查仅取决于已存在的状态，以及在同一个事务中的状态。如果我们试图在另一个事务中移除一个名字，这将对我们毫无影响。即便出现冲突，其中一个事务会再次尝试。

为我们写代码的代码

Clojure 作为一门 Lisp 语言，一样提供了强力的宏系统。宏允许对重复模板代码块延迟求值，有着广泛的用途。宏会将代码如同数据处理，而不是求值。这允许我们如同其他数据结构一样操作代码树。

宏操作在编译器之前执行，编译器看到的是宏操作的执行结果。因为这一级是间接的，我们对宏推断起来很费劲，因此，如果函数能做的事，最好不要用宏来实现。

不过，如果合理使用宏，还是有必要了解用法。本书中极少使用宏，因此，我们对其语法点到为止。

我们通过一个使用宏的具体例子来说明它和我们之前见到的常规代码的区别。我们试想一下，如果有个 session，在用户登录之后包含账户信息。假如在会话中存在用户，我们可能需要从中加载一些内容。

```
(def session (atom {:user "Bob"}))

(defn load-content []
 (if (:user @session)
   "some content"
   "please log in"))
```

这就行了，但对于每次都要写下 if 去判断显得有些枯燥。由于业务逻辑会比较类似，我们可以将这个函数如下模板化：

```
(defmacro defprivate [name & body]
 `(defn ~(symbol name) []
   (if (:user @session)
     (do ~@body)
     "please log in")))
```

上面的宏通过 defmacro 表达式定义。defn 和 defmacro 最大的区别在于，一般在传入 defmacro 的参数并不会被求值。

我们可以通过波浪线 (~) 前缀对参数求值，使用“~”符号表示我们希望通过名字

的值替换此处。这称为引述（Quote）。

在(do ~@body)这段代码中，“~@”符号称为解引述拼接（Unquote Splicing）。这种符号用于处理一个序列。序列中的内容在拼接时会被合并进入输出表达式。这种情况下序列体代表函数体。需要强调一下，这个传入的代码块必须放在 do 函数内部，因为 if 函数只支持两个参数。

“`”符号表达了视图将接下来的一个执行语句序列作为数据处理。这跟解引用正好相反，并将调用作为符号引用。

我之前提到过，宏在编译期之前执行。要想知道宏在编译器展开的样子，我们调用 macroexpand-1 就能看到。

```
(macroexpand-1 '(defprivate foo (println "bar")))

(clojure.core/defn foo []
  (if (:user (clojure.core/deref user/session))
    (do (println "bar"))
    "please log in"))
```

你可以看到，(defprivate foo (println "bar"))通过函数定义重写，其中已经有了 if 判断。结果代码就是编译器看到的样子，这也是假如我们要通过手写实现的内容。接下来，我们可以通过使用宏简单定义一个 private 函数，让其为我们自动检查。

```
(defprivate foo (println "bar"))
```

以上示例代码看起来可能不太直观，不过充分说明可以通过代码实现简单重复模板。这便可以创造符号表达式，来表达你面对的问题域中最直接的表述。

REPL

使用 Clojure 还有个东西不得不提，那就是“读取-求值-输出”循环（REPL，Read-evaluate-print Loop）。用很多语言在写代码时，我们都是运行整个项目。Clojure 的绝大部分开发过程都是在和 REPL 打交道。这种模式最大的好处就是写下的每段代码都可以马上看到运行效果。

在很多普通的应用中，经常需要完成某种特殊阶段之后，你才能添加更多功能。例如，一位用户登录后，可以查询后端的一些数据，接下来，你需要编写函数去格式化显示数据。通过 REPL，你可以随时获取应用序状态，甚至在数据加载之后，交互

式编写显示逻辑，而不用在每次更改之后，重新加载应用程序并恢复状态。

我发现这个方法在开发时非常令人满意，因为对程序添加代码或修改之后，可以立即获得反馈。你很容易做各种尝试，从而了解运转情况。它鼓励实验并在实践中重构代码，这有助于你写出更好、更整洁的代码。

综述

我们的 Clojure 基础之旅到此结束。虽然我们只接触整个语言非常小的一部分，但如果你理解了前面的例子，那么你通读书中其他的代码应该已无障碍。在搭建好开发环境并运行之后，最好将这里提及的例子尽数在 REPL 中把玩一番，直到你感觉进入状态再继续。

附录 3

面向文档的数据库访问

一个 SQL 数据库可能并不总是很适合你的程序，很多应用并不需要关系实例。如果应用程序需要持久化层去存储，且需要检索记录，那么文档存储可能更适合。

选择正确的数据库

选择基于文档的数据库主要从三个方面考虑。这就是由 CAP 定理[①]所指的一致性、可用性以及分区容忍性。由于这些目的本身相互矛盾，当面临选择数据存储时，必须从中挑选对你最有价值的两个。

一致性

当拥有一致性，每个客户端都拥有同样的数据视图。当拥有大量节点的数据库集群时，这方面将发挥作用。在一致性数据库中，保障每个节点拥有同样的数据视图。

有些数据库，比如 CouchDB[②]，提供最终一致性。这意味着，集群中的每个节点都是自治的（self-consistent），不保证提供最新的数据。

---

① http://en.wikipedia.org/wiki/CAP_theorem
② http://couchdb.apache.org/

可用性

可用性指的是数据库没有全局锁。一个客户端连接到任何节点都能随意读写。不过，数据终究还是能保障通过集群传播。此原则的负面效果是客户端不是每次都能获取最新的数据。

CouchDB 用的是这种模型，从而提供了高可靠性集群。需要注意，集群通常拥有奇数个节点。这是为了 CouchDB 在出现记录冲突时，方便判定取舍。这种情况用于两个不同的客户端，连接不同的节点并更新了同一条数据。在这种情况下，其中一个客户端会被以修订告终。

分区容忍性

支持分区容忍性的数据库能在跨越物理网络的条件下良好工作。这意味着，即便集群中发生了严重的网络故障，当网络恢复后，节点间也能自动同步。

使用 CouchDB

CouchDB 重视可用性和分区容忍性。这使其非常合适去创建实现几乎不受限的高吞吐集群。

在本节中，我们通过在 Clojure 中如何使用 CouchDB 实现一些基本任务，比如对文档的存储、检索以及删除。

运行以下示例的前提是要么在本地搭建 CouchDB，要么使用免费的 CouchDB 服务，比如 Iris Couch[①]。搭建好数据库之后，访问 http://hostname:5984/_utils，使用其 Web 界面新建一张名为 clutchtest 的数据表。

Clutch 库

用 Clojure 访问 CouchDB 最简单的方式是使用 Clutch 库[②]。Clutch 提供了非常简

① http://www.iriscouch.com/
② https://github.com/clojure-clutch/clutch

单和直观的接口。要使用这个库之前，我们必须在当前项目中添加依赖项，写作本书时的最新版是[com.ashafa/clutch "0.4.0-RC1"]。

连接到数据库

要使用 clutch，我们必须在命名空间声明处引用。

```
(:require [com.ashafa.clutch :as couch])
```

接下来，我们定义连接 URL。由于 CouchDB 通过 HTTP 访问，我们的 URL 可以使简单字符串指定数据库地址。

```
(def db "http://localhost:5984/clutchtest")
```

我们应该还添加此 URL 的验证，可以直接写在连接字符串里面。

```
(def db "http://user:pass@localhost:5984/clutchtest")
```

或者使用 URL 库[①]来创建 URL，然后作为 map 附加验证信息。

```
(def db (assoc (cemerick.url/url "https://localhost:5984/" "clutchtest")
               :username "user"
               :password "pass"))
```

现在，我们已经创建了连接，接下来看看如何在数据库中存储文档。

存储文档

与数据库的所有交互必须插入 with-db 宏。这个宏会确保任务完成之后，合理关闭连接。

在数据库存储文档，我们可以调用 put-document 函数，并传递给它一个表示文档的 map。

```
(couch/with-db db
  (couch/put-document {:foo "bar"}))
```

① https://github.com/cemerick/url

上面的代码允许我们在数据库中创建一个新文档，并为其生成并分配了一个随机 ID。要为保存的文档指定 ID，可以在传入的 map 参数中添加:id 键。

```
(couch/with-db db
  (couch/put-document
    {:_id "user" :username "foo" :pass "$dfsdf#23434"}))
```

当需要更新一条已有的文档，我们还必须加载对当前文档的修订 map。例如，如果已经插入一条 user 文档到数据库，我们现在需要在更新时通过:rev 指定修订版。

```
(couch/with-db db
  (couch/put-document
    {:_id "user" :_rev "<revision number>" :username "foo" :pass "$dfsdf#23434"}))
```

当从数据库检索文档时，会返回:_id 和:_rev 键，因此当确认希望保留时，可以再次保存。现在，我们看看如何从数据库获取文档。

检索单个文档

使用 get-document 函数能从数据库找回文档，其接受的参数字符串代表要找回的文档 ID。

```
(couch/with-db db
  (couch/get-document "user"))
```

我们还可以查询多个文档，在一个 with-db 语句中，插入多条查询语句。例如，如果希望查询 user，设置一个新用户名，并保存文档，我们可以如下操作：

```
(couch/with-db db
  (let [doc (couch/get-document "user")]
    (couch/put-document
      (assoc doc :username "bar")))
  (println (couch/get-document "user")))
```

检索多个文档

有时候，我们需要从数据库批处理取回多个文档。Clutch 为我们提供一个函数用

来处理这种需求，名为 all-documents。

```
(couch/with-db db
  (couch/all-documents))
```

前面的调用会返回指定数据库的所有 ID 及修订。还可以设置:include_docs 键从数据库获取完整的文档。

```
(couch/with-db db
  (couch/all-documents {:include_docs true}))
```

此外，我们可以约束检索内容来实现批量检索，设置:keys 键指定需要的一组 ID，如下所示。

```
(couch/with-db db
  (couch/all-documents
    {:include_docs true}
    {:keys ["doc1" "doc2" "doc3"]}))
```

要通过 CouchDB 实现更复杂的选定，你可能需要基于应用需求定制视图，通过过滤并返回文档。视图类似于关系型数据库的存储过程。

删除文档

最后，我们使用 delete-document 函数删除文档。它接受字符串文档 ID，并从数据库删除文档。

```
(couch/with-db db
  (couch/delete-document "user"))
```

这就是通过 Clojure 使用 CouchDB 的全部内容。Clutch 使得从数据库存储和检索变得简单，直接通过 rich-views 可以将更复杂的功能添加到数据库。现在我们来看看使用 Monger[①]库访问 MongoDB[②]。

使用 MongoDB

MongoDB 是另一个受欢迎的面向文档数据库。不像 CouchDB，它以支持一致

① http://www.mongodb.org/
② http://clojuremongodb.info/

性和分区容忍为主要目标。如果你不介意有全局锁，那么 MongoDB 会是非常棒的选择。

连接数据库

我们使用 Monger 访问 MongoDB。Monger 提供 Clojure 编程风格接口来访问数据库，并全面支持 MongoDB 2.2 以上版本数据库特性。和 Clutch 的情况一样，我们可以使用原生 Clojure 数据结构而不用操心如何转换为 MongoDB/BSON 格式。最后，Monger 的默认配置是强调安全性和一致性。

调用 monger.core/connect!就能很容易连接数据库。在不提供参数的情况下，connect!会试图以默认端口连接本地数据库接口。我们还可以提供包含:host 和:port 键的 map 来访问，或通过 mongo- options 配置连接。我们接下来看看具体使用：

```
(ns mongo-example.core
  (:require [monger.core :as [m]])
  (:import
    org.bson.types.ObjectId
    [com.mongodb MongoOptions]))

;;connects to a local instance
(m/connect!)
;;connect to myhost.com on port 5001
(m/connect! {:host "myhost.com" :port 5001})

;;connect using custom options
(m/connect! (m/server-address "127.0.0.1" 27017)
            (m/mongo-options
              :threads-allowed-to-block-for-connection-multiplier 300))
```

我们还可以使用默认数据库配置*mongodb-database*，并使用 set-db!函数对其进行设置，如下所示。

```
(defn connect! [& [params]]
  ((partial monger.core/connect!) params)
  (monger.core/set-db! (monger.core/get-db "local")))
```

使用 set-db!配置数据库将隐式影响此后的查询操作。

绝大多数关于数据库的操作由 monger.collection 命名空间提供。这里介绍插入、选择、更新和删除记录的操作，我们来分别看看。

插入记录

我们使用 insert 函数在数据库插入一条新记录。函数接受数据集的名称，以及一个以字符串为值的 map 作为要插入的文档。

```
(monger.collection/insert "users" { :first_name "John" :last_name "Doe" })
```

函数返回一个存储结果，通过使用 monger.result/ok?检查返回状态。当写入成功，monger.result/ok?会返回 true。

如果需要对文档指定 ID，我们需要通过使用 org.bson.types.ObjectId 生成。

```
(monger.collection/insert "users" { :first_name "John" :last_name "Doe" })

(monger.collection/insert
  "users"
  { :_id (ObjectId.) :first_name "John" :last_name "Doe" })
```

接下来，我们使用 insert-and-return 函数插入记录。除了以 map 返回插入的文档，其操作与 insert 类似。

```
(monger.collection/insert-and-return "users"
  { :_id (ObjectId.) :first_name "John" :last_name "Lennon" })
```

我们还能使用 insert-batch 函数批量插入。这个函数接受一个数据集名，以及一个表示文档 map 的序列。

```
(monger.collection/insert-batch
  "users"
  [{ :first_name "John" :last_name "Doe" }
   { :first_name "Jane" :last_name "Smith" }])
```

选择记录

Monger 提供几个函数用来搜索记录，并以 Clojure 的 map 返回。这些函数是 find-maps、find-one-as-map 以及 find-map-by-id。

find-maps 函数可以在数据集中使用包含键值的 map 查询文档，并会返回包含指定

键的对象。如果不指定任何参数，将返回所有文档。

```
(monger.collection/find-maps "users" {:first_name "John"})
```

find-one-as-map 函数仅返回满足条件的唯一结果。

```
(monger.collection/find-one-as-map "users"
  { :first_name "John"})
```

最后，find-map-by-id 函数接受一个对象 ID 作为搜索参数。

```
(monger.collection/find-map-by-id "users"
  (ObjectId. "514f455d03642f52431b5bfe"))
```

甚至还允许使用标准 MonogDB 查询操作完成查询，如下所示。

```
(monger.collection/find-maps "products" { :price { "$gt" 300 "$lte" 5000 } })
```

更新记录

我们通过使用 update 函数更新记录，还可以增加:upsert true 来表示如果更新的记录不存在，改为插入操作。

```
(update "users" { :first_name "John" :last_name "Doe" })
;;update existing or insert a new record
(update "users" { :first_name "John" :last_name "Doe" } :upsert true)
```

删除记录

最后，我们还可以使用 remove 函数从数据库删除文档。如果我们不指定任何条件，会移除所有的文档。

```
;;remove ALL documents
(monger.collection/remove "users")

;;remove documents with the specified key
(monger.collection/remove "users" { :language "English" })
```

如你所见，使用面向文档数据库相当简单。根据应用程序的需要，你可以使用文档型而不是关系型数据库，或者将两者适当结合。无论你选择何种方式，Clojure 都能兼顾。

Clojure Web 开发实战

Web Development with Clojure

Clojure提供了基于JVM的丰富的基础设施，以及强大的函数式语言表达，它性能极佳且兼顾开发高效，你只需要在Web应用的开发中将这些优势发挥出来。

本书从Clojure编程理论出发，最终落实到具体的开发实践过程。你会通过Clojure这种强力的语言来处理整个Web程序的每一个细节。

通过本书，你将能够：

- 了解使用Clojure进行Web开发的全过程，尝试这门语言时下最新的工具、库以及最佳实践；
- 学习通过Light Table和Eclipse两个开发环境来开发Clojure应用；
- 对流行的Ring/Compojure栈有深刻的认识，并且学会使用Liberator库快速搭建RESTful服务；
- 了解如何通过ClojureScript让服务端、客户端工作在同一种语言上；
- 体验开发Web程序的关键部件，包括通过多种途径访问数据库；
- 创建一个简单的留言簿程序以及一个为用户提供资源的应用；
- 开发一个功能丰富的相册网站，从前期构思、打包直到最终部署上线。

通过本书，参考实例、循序渐进地学习，您将深刻领悟到使用这种强大且丰富的工具来打造现代Web应用程序的全过程。本书对任何想开发Web应用的人来说，都是“根本就停不下来”。如果你已经对Clojure有一定了解，你一定可以活学活用，将其能量充分发挥。即便你初次接触Clojure，本书也足以让你游刃有余地使用它。

作者简介

Dmitri Sotnikov专注开发Web应用已近十年，已开发并持续维护几个大型的Clojure库（类似Luminus框架）。

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

分类建议：计算机／程序设计／Web开发
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-39893-2

ISBN 978-7-115-39893-2

定价：45.00 元